Assignment 2
CSCI 415
Perry, the Python Proxy

Cesar Ramirez, Josh Tan

October 7, 2013

Overview

Our proxy is implemented in Python 3 and consists of four main classes: `ProxyConn`, `ClientRequest`, `HttpRequest`, and `Cache`. The `ProxyConn` is responsible for managing and executing each `ClientRequest`, which contains the `HttpRequest` made by the client. The responses for client requests are maintained in a `Cache` object, which is implemented as a Least Recently Used (LRU) cache. We use the low-level Python `socket` module for both client and server socket connections. Multi-threading is accomplished using the low-level `threading` module. To synchronize the different threads during manipulation of the cache and log file, we utilize the `threading.Lock` class.

The program begins by invoking the `start_server()` method, which creates the server socket and begins listening on it for incoming client connections. Whenever a client connection is established, this connection is sent to a `ProxyConn` object in a separate thread. The server will always have n+1 running threads, where n is the number of active connections.

Main Classes

ProxyConn

The `ProxyConn` object is the entire lifecycle of a client requesting a website through this proxy. This object will receive the socket connection of the client and create a `ClientRequest` object to contain it. After initializing and executing the `ClientRequest`, the client connection is closed.

ClientRequest

The `ClientRequest` object creates an `HttpRequest` object from the data provided. Based on the parameters that are passed to it, it may strip the cache and user-agent related headers from the request. The latter is done to increase the anonymity of the request. The most important method in this class is the `execute()` method, which is where the client-server data relay occurs. This method also interacts with the proxy cache and performs the logging activities.

If the HTTP request for the client request has been seen before, the corresponding response will be sent back to the client directly from the cache. This response will also be moved to the front of the cache. If the request is new, the `ClientRequest` will instead connect to the remote server and relay the server response back to the client, while also placing this response into the cache for future use.

HttpRequest

A `HttpRequest` is simply an abstraction of the HTTP request from the client. The goal of this class is to provide a way to easily manage and handle the request line as well as the request headers. This object is constructed from the raw text from the client, utilizing member attributes and a header dictionary to store the parsed data.

Cache

Our cache follows a Least-Recently-Used model, with all major operations running in constant time (under optimal circumstances). It consists of a queue, implemented as a doubly linked list, as well as a dictionary mapping response hash IDs to the entries. Each entry is moved to the beginning of the queue during creation or cache retrieval. When the cache is full (i.e., has reached the defined maximum size), the entries that are at the end (that have been least-recently used) are removed, until enough space has been freed. Furthermore, the entry dictionary is used to achieve most reading operations in constant time.

We utilize a cache-wide lock that ensure all read/writes to the cache are thread-safe. The maximum cache size is specified in terms of bytes, using a global variable. Within memory, we keep a reference to each entry

of the cache. However, the corresponding response data itself is stored in a file on disk. These files are uniquely identified by the entry key, which is a MD5 hash of the request. To increase efficiency, the cache is only locked during the `insert()` method, when entries are re-organized or deleted to make room for a new request/response entry.