Proxy Homework

By Linxia GONG and Yiyi ZHANG

Task

Design and implementation of a proactive, client-side HTTP cache,requirement:

- Implement a cache that speeds up the access to content
- Parallelize the downloads in order to increase the bandwidth available

Realization

1. Basic function

Receive and forward to the remote webserver the requests from user.

Here we utilise some libraries of python to facilitate the bulid of proxy: *socket*, *ssl*, *httplib*, *urlparse*, *BaseHTTPServer*, *SocketServer*, *cStringIO*, *subprocess*, *HTMLParser*.

With these libraries, we can easily establish a webserver ourselves.

```
def main(HandlerClass=ProxyRequestHandler, ServerClass=ThreadingHTTPServer,
protocol='HTTP/1.1'):

    HandlerClass.protocol_version = protocol
    httpd = ServerClass(server_address, HandlerClass)

    sa = httpd.socket.getsockname()
    print "Serving HTTP Proxy on", sa[0], "port", sa[1], "..."
    httpd.serve_forever()
```

2. HTTPS connection

The **`httplib`** module has already included the methods to deal with HTTP and HTTPS protocols from a client side. What we need to do here is to tell apart the two kinds of requests.

```
try:
    origin = (scheme, netloc)
    if not origin in self.tls.conns:
        if scheme == 'https':
            self.tls.conns[origin] = httplib.HTTPSConnection(netloc,
timeout=self.timeout)
        else:
            self.tls.conns[origin] = httplib.HTTPConnection(netloc,
timeout=self.timeout)
    conn = self.tls.conns[origin]
    conn.request(self.command, path, req_body, dict(req_headers))
    res = conn.getresponse()
    res_body = res.read()
except Exception as e:
    if origin in self.tls.conns:
    del self.tls.conns[origin]
    self.send_error(502)
    return
```

3. Multi-tasks

It's easy to realize multi-tasks requirement in **SocketServer** module. There are two ways to solve asynchronous assignments by creating a separate process or thread to handle each request, using the ForkingMixIn or ThreadingMixIn mix-in classes.

```
class ThreadingHTTPServer(ThreadingMixIn, HTTPServer):
    # Change HTTP server into a threading one
    address_family = socket.AF_INET6
    daemon_threads = True

    def handle_error(self, request, client_address):
        ...
```

4. Cache

1.In the proxy, we make a hash table as our website cache. For every cache, we define a class to normalize it.

```
cache = {}

class cache_content():
    # define the class of every cache
    def __init__(self):
        self.protocol_version=""
        self.res_status= 0
        self.res_reason=""
        self.res_headers=""
        self.res_body=""
```

2.Key for cache table

We catch the pathname from every request and set it as the key to cache table.

3.Storage of cache

Everytime when the proxy connects with the remote webserver, it will create a new object of cache_content class, and save the necessary information and the html code into the cache table.

```
classtest = cache_content()
                cache[key]= classtest
                cache[key].protocol_version = self.protocol_version
                cache[key].res_status = res.status
                cache[key].res_reason = res.reason
                cache[key].res_headers = res_headers.headers
                cache[key].res_body = res_body
```

4.Retrieve data from cache

Everytime proxy gets a request from user, it will analyse if the pathname is already in the cache table or not.

```
key = req.path
for k in cache.keys():
    if k == key:
        flag = "cache"
```

If it's existed, then the proxy will load data from cache instead of connecting to the remote webserver.

```
self.wfile.write("%s %d %s\r\n" % (cache[key].protocol_version,
cache[key].res_status, cache[key].res_reason))

for line in cache[key].res_headers:
    self.wfile.write(line)
self.end_headers()
self.wfile.write(cache[key].res_body)
self.wfile.flush()
flag = 'server'
```