

Relazione progetto “Reti di Calcolatori” A.A. 2017/2018



Nome: Matteo Mauro
Matricola: 5971842

INDICE

1.0 Analisi funzionale: descrizione delle caratteristiche implementate	3
2.0 Analisi tecnica: descrizione delle scelte implementative	5
2.1 Handlers	5
2.1.1 Classe: MyAbstractHTTPHandler	5
2.1.2 Classe: HTTPHandler1_0	5
2.1.3 Classi: GET_HTTPHandler1_0, POST_HTTPHandler1_0, HEAD_HTTPHandler1_0	6
2.1.4 Classe: CompositeHTTPHandler1_0	8
2.1.5 Classe: HTTPHandler1_1	8
2.1.6 Classe: PUT_HTTPHandler1_1	9
2.1.7 Classe: DELETE_HTTPHandler1_1	9
2.1.8 Classe: HTTPProxyHandler	10
2.2 Streams	11
2.2.1 Classe: MyHTTPInputStream	11
2.2.2 Classe: MyHTTPOutputStream	11
2.2.3 Classe: HTTPMessageParser	11
2.3 Classe: MyHTTPServer	14
2.4 Classe: HandlersPool	15
2.5 Classe: HTTPReplyThread	16
2.6 Classe: MyHTTPRequest / MyHTTPReply	17
3.0 Schema UML	18
4.0 Test	19
4.1 [GET/HEAD/POST]_HTTPHandler[1.0/1.1]Test	19
4.2 HTTPMessageParserTest	20
4.3 MyHTTPInputStreamTest	21
5.0 Guida all'utilizzo del programma e degli script Python	22

1.0 Analisi funzionale: descrizione delle caratteristiche implementate

Il programma realizza tutte le caratteristiche richieste dal progetto, implementando un server multi-thread che risponde alle richieste HTTP semanticamente corrette con una risposta HTTP conforme alle specifiche.

Il server può gestire le seguenti categorie di richieste:

- HTTP/1.0: GET, POST, HEAD
- HTTP/1.1: GET, POST, HEAD, PUT, DELETE

Inoltre consente di usufruire di richieste di connessioni persistenti e non persistenti: se il campo "Connection" non è esplicitato nella richiesta, il comportamento di default è non persistente per un metodo HTTP/1.0, altrimenti è persistente per HTTP/1.1 (come indicato qui, **sezione 8.1.2:** <https://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html>).

E' possibile scegliere quale tipologia di richiesta poter gestire semplicemente inserendo gli handler desiderati (come dimostrazione la classe "MyHTTPFactory" li inserisce tutti); per evitare la proliferazione di troppe istanze uguali degli stessi Handlers, è stata realizzata la classe "HandlersPool": il suo scopo è contenere l'insieme degli handlers istanziati dal server ed agire come un monitor, mettendo a disposizione i propri servizi a tutti i thread che devono gestire le richieste dei rispettivi clients. HandlersPool si occupa anche dell'aggiornamento costante del file di log (per le motiviamo di tali scelte implementative leggere il paragrafo "2.4 Classe: HandlersPool").

L'utente può decidere da quale directory root eseguire la ricerca delle risorse richieste dai clients: se non specificata la ricerca sarà circoscritta alla cartella "www/" presente nella root del file system, altrimenti il percorso scelto dall'utente (e.g. "anotherFolder/") sarà identificato a partire da "www/anotherFolder/".

Attenzione: creare nella directory root la cartella "www/" e qualsiasi altra sottocartella desiderata, altrimenti il programma non sarà in grado di rintracciare eventuali file richiesti e restituirà sempre un "404 File Not Found". Usare eventualmente quella fornita con il progetto.

```
iMac-di-Matteo:~ matteomauro$ cd /
iMac-di-Matteo:/ matteomauro$ pwd
/
iMac-di-Matteo:/ matteomauro$ ls
Applications      installer.failurerequests
Library           model
Network          net
System           opt
Users            private
Volumes          sbin
bin              tmp
cores            usr
dev              var
etc              www
home
iMac-di-Matteo:/ matteomauro$ cd www
iMac-di-Matteo:www matteomauro$ ls
anotherFolder  index.html
iMac-di-Matteo:www matteomauro$
```

Ogni handler può essere istanziato indipendentemente con lo scopo di gestire le risorse a partire da una specifica cartella e/o per conto di uno specifico Host: in quest'ultimo caso l'handler sarà incaricato di rispondere esclusivamente alle richieste che riportano lo stesso Host nell'intestazione. A tal proposito, ogni richiesta HTTP/1.1 deve necessariamente specificare un parametro "Host" (al contrario di HTTP/1.0), pena l'immediata respinta dell'handler (come descritto nella sezione "**Internet address conservation**": <http://www8.org/w8-papers/5c-protocols/key/key.html>). Se un handler è stato creato con Host uguale a null, allora di default può rispondere a qualsiasi request ricevuta senza restrizioni.

Gli script python forniti realizzano una serie di invii di richieste per ogni metodo:

- il file "reqNonPersistent.py" contiene delle funzioni che inviano richieste non persistenti;
- il file "reqPersistent.py" invia 4 request in pipelining, le prime 3 sono richieste GET persistenti, l'ultima è una richiesta POST che chiude la connessione (i metodi sono scelti in maniera casuale come esempio esplicativo);
- il file "reqProxy.py" invia 2 request GET per ricevere la index.html di "www.google.it", il sever riceve i messaggi ed inoltra il primo attraverso l'HTTPProxyHandler, salvando il contenuto della risposta in locale.

L'ultima menzione riguarda il file di log: il file log.txt che viene aggiornato dall'applicazione viene caricato attraverso il metodo getResource(), che consente di accedere alle risorse salvate nella cartella bin (in caso di deployment dell'applicazione in formato .jar).

Per vedere esempi di utilizzo del programma, leggere la sezione "5.0 Guida all'utilizzo del programma e degli script Python".

ATTENZIONE: prima di avviare i test leggere attentamente la sezione 4.0, specialmente la parte che riguarda l'insieme delle risorse utilizzate dai test (cartella www/ e file necessari ecc).

ATTENZIONE: dal progetto è stata eliminata la classe contenente il metodo *main()* prima della consegna.

2.0 Analisi tecnica: descrizione delle scelte implementative

La seguente trattazione è strutturata in base ai package di appartenenza delle classi discusse.

Handlers -> Streams -> Server e classi restanti

Quando ritenuto opportuno saranno presenti delle immagini esplicative della funzionalità realizzata e descritta.

2.1 Handlers

2.1.1 Classe: MyAbstractHTTPHandler

La classe "MyAbstractHTTPHandler" è stata progettata seguendo il "Template pattern" per rifattorizzare le operazioni comuni all'interno del metodo handle(). Tali passaggi preliminari, realizzati indipendentemente dall'handler, sono:

- il controllo del metodo specificato (GET, POST ecc) -> isMyMethod();
- il controllo della versione del protocollo (HTTP/1.0 non può gestire 1.1, ok il viceversa) -> isMyVersion();
- il controllo dell'host (se l'handler non è responsabile per rispondere all'host specificato nella request, allora viene subito respinta) -> isMyHost().

Se i suddetti controlli sono superati con successo, allora viene richiamato il metodo doHandle(), ovvero l'"hook" che verrà implementato specificatamente da ogni handler concreto.

La classe contiene gli attributi privati per salvare l'host e la directory root possibilmente passati come argomento (rispettivamente "String host" e "File directoryRoot").

2.1.2 Classe: HTTPHandler1_0

Per semplificare l'interazione nei confronti degli handlers, è stata realizzata la classe astratta HTTPHandler1_0 sulla base del "Composite pattern": in tal modo il server deve semplicemente passare la request all'oggetto composto, il quale sarà in grado di rintracciare l'handler competente (se presente) e rispondere correttamente; in tal modo è possibile decidere quali metodi (GET, POST...) gestire scegliendo quali handler istanziare e aggiungere all'aggregato.

La struttura è stata opportunamente divisa tra gli handler che sono compatibili con la versione 1.0 e 1.1, così da poter limitare (se ritenuto necessario) la versione che il

server deve poter gestire e risulta pertanto facilmente scalabile con l'estensione di altre versioni, dal momento che le operazioni comuni sono già rifattorizzate in `MyAbstractHTTPHandler`.

2.1.3 Classi: `GET_HTTPHandler1_0`, `POST_HTTPHandler1_0`, `HEAD_HTTPHandler1_0`

Premessa 1: in conformità con il "Composite pattern" che è stato adottato, le seguenti classi descritte rappresentano le "foglie", pertanto il metodo `add()` lancia un'eccezione con un relativo messaggio d'errore.

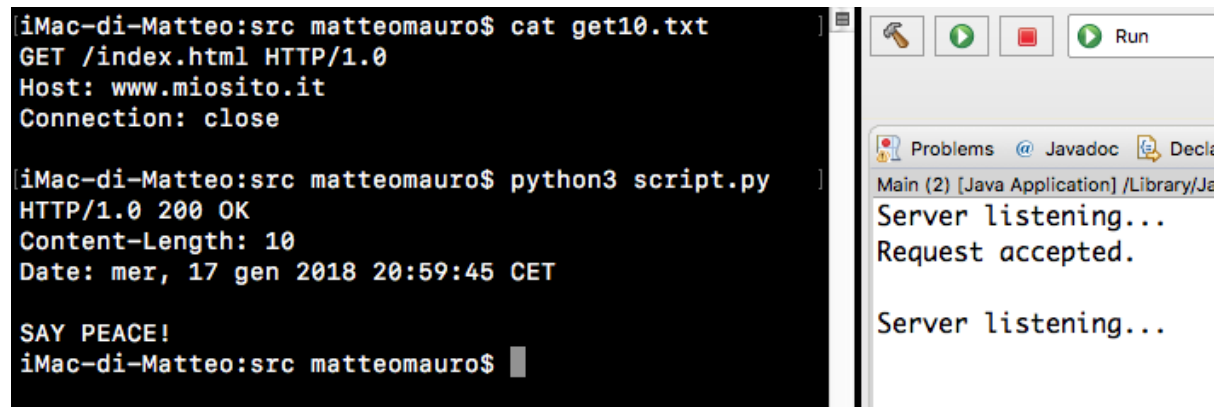
Premessa 2: ogni metodo verifica la presenza del file esplicitato nella request line, se il file non è presente viene restituita una reply "404 File Not Found" opportunamente costruita.

GET 1.0

Il metodo `doHandle()` di `GET` verifica che la risorsa richiesta (per esempio un file "index.html") sia presente nella `directoryRoot` specificata: se esiste richiama la funzione privata `readFile()` per ottenere una stringa rappresentativa del contenuto (che sarà usato come content body della reply), altrimenti costruisce una risposta con status line "404 File Not Found". Ogni `HTTPReply` inviata avrà (indipendentemente dal metodo) le seguenti header lines:

- Content-Length: <value> -> se il body è vuoto sarà impostato a 0;
- Date: <EEE, dd MMM yyyy HH:mm:ss z> -> la classe `MyHTTPUtility`, attraverso il metodo statico `getDefaultHeaderLines()`, può fornire una rappresentazione affidabile della data di creazione della reply;

Altri parametri possono essere inclusi in altri metodi se ritenuto necessario.



```
iMac-di-Matteo:src matteomauro$ cat get10.txt
GET /index.html HTTP/1.0
Host: www.miosito.it
Connection: close

iMac-di-Matteo:src matteomauro$ python3 script.py
HTTP/1.0 200 OK
Content-Length: 10
Date: mer, 17 gen 2018 20:59:45 CET

SAY PEACE!
iMac-di-Matteo:src matteomauro$
```

Problems @ Javadoc Decla

Main (2) [Java Application] /Library/Ja

Server listening...
Request accepted.

Server listening...

Esempio di richiesta/risposta: il file get10.txt contiene la richiesta semanticamente corretta (visualizzata con "cat get10.txt"), che viene utilizzata nello script python per essere inviata al server Java: quest'ultimo è sempre attivo, quando riceve una richiesta la delega ad un thread e riprende l'ascolto. Infine la reply viene inviata al client che la mostra a schermo.

POST 1.0

In generale, il metodo POST è utilizzato per trasportare dati all'interno del body, con l'eventuale scopo di aggiornare un record presente sul server. Per simulare questo comportamento viene verificato che il file da aggiornare (specificato nella request line) sia presente sul server (analogamente al metodo GET), infine (non essendo presente un effettivo record da aggiornare nel file) i dati ricevuti vengono semplicemente stampati su standard output.

Nel contesto di questo programma, per rendere più netta la differenza tra il metodo POST e gli altri, è stata inserita nella header section della reply la linea "Location:" che specifica il percorso della risorsa sul sever secondo questo schema:

Location: http://<host>/<directoryRoot>

esempio-> Location: http://www.miosito.it/www/randomFolder/index.html

HEAD 1.0

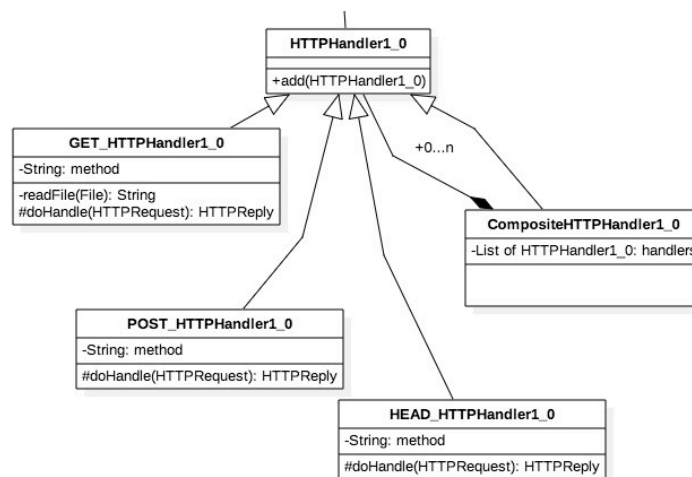
L'implementazione di HEAD riassume il comportamento di GET senza restituire il body (Content-Length: 0) e aggiunge la Location come suggerito nel metodo POST.

2.1.4 Classe: CompositeHTTPHandler1_0

CompositeHTTPHandler1_0 rappresenta l'oggetto che potrà essere composto esclusivamente da istanze di tipo HTTPHandler1_0: ciò è garantito con l'introduzione del generic <HTTPHandler1_0> al campo privato LinkedList. In conformità con il Composite pattern, l'aggregato può aggiungere un numero indefinito di handlers attraverso il metodo add().

Il metodo handle() itera la lista di handlers correntemente presenti nell'aggregato: dall'esterno il comportamento è simile a quello del metodo handle() di un qualsiasi handler "foglia", dal momento che restituisce una reply se l'handler può gestire la request oppure null in caso negativo; ovviamente il composto non può eseguire i metodi isMyMethod() e doHandle() (coerenti solo con handler foglie), perciò essi lanciano un'eccezione.

Rappresentazione UML della struttura "Composite" (specifica della versione 1.0, la 1.1 è analoga, vedere l'immagine "UML.jpeg" per lo schema completo)



2.1.5 Classe: HTTPHandler1_1

Considerata l'evidente somiglianza di implementazione tra le due strutture Composite per la versione 1.0 e 1.1, mi soffermo solo sulla descrizione delle differenze principali.

La classe HTTPHandler1_1 introduce un nuovo livello di "Template pattern": dal momento che (come accennato nell'Analisi funzionale) ogni HTTPHandler1_1 esige che il parametro "Host" sia presente all'interno della request ricevuta, tale controllo è stato demandato a questa classe all'interno del metodo doHandle() (che ricordo era esso stesso l'"hook" della superclasse); se tale controllo viene superato allora viene

richiamato il metodo `doHandle1_1()` da ogni classe concreta (rappresenta l'“hook” del template).

Ogni request con versione 1.0 deve essere gestita correttamente da ogni versione superiore, quindi il metodo `isMyVersion()` della classe `HTTPHandler1_1` è stato modificato per essere conforme a questa direttiva.

Dato che `GET1.1`, `POST1.1` e `HEAD1.1` e `CompositeHTTPHandler1_1` sono praticamente uguali alle versioni 1.0 nella struttura e implementazione, si analizzano solo i metodi `PUT` e `DELETE`.

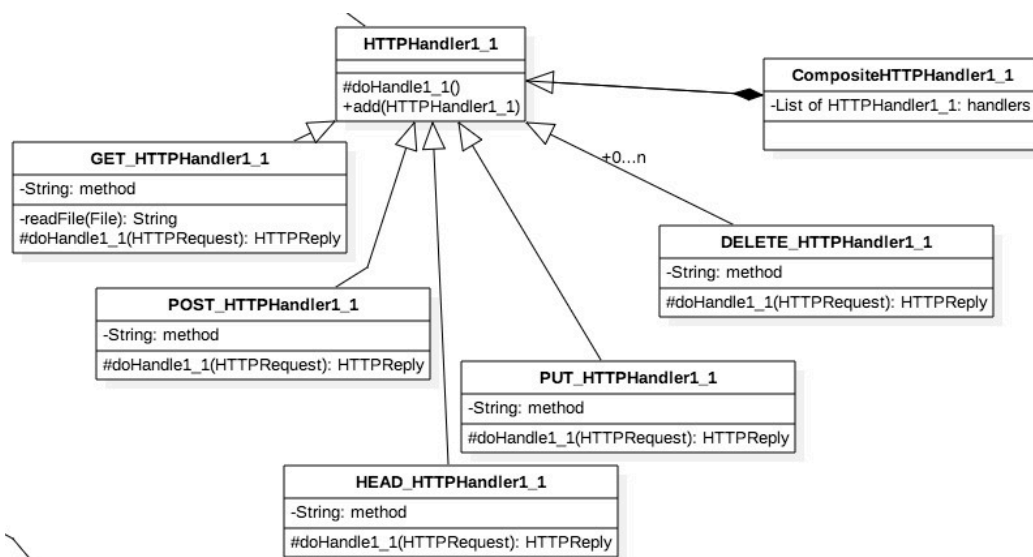
2.1.6 Classe: `PUT_HTTPHandler1_1`

Il metodo `PUT` consente di creare sul file system del server un nuovo file, il cui nome è specificato nella request line e il contenuto è quello trasmesso nel body. Se il file è già presente nella `directoryRoot` specificata, esso viene sovrascritto.

Se la procedura ha esito positivo viene inviata una reply “200 OK”, altrimenti viene inviata una reply “409 Conflict” per ogni `IOException` rilevata.

2.1.7 Classe: `DELETE_HTTPHandler1_1`

Il metodo `DELETE` consente di cancellare permanentemente il file specificato nella request. Inizialmente si verifica che il file esista (in caso negativo viene inviata una reply “404 File Not Found”), in seguito si esegue la cancellazione attraverso il metodo `delete()` della classe `File` di Java: se dopo la cancellazione il file non risulta presente (`File.exists()`) allora è stato cancellato con successo e viene inviato un “200 OK”, ma se il file risulta ancora esistente allora è avvenuto un conflitto e si invia la reply “409 Conflict”.



Rappresentazione struttura Composite versione 1.1

2.1.8 Classe: HTTPProxyHandler

Il proxy è stato realizzato introducendo un attributo privato di tipo `Map<String, String>` di nome `cachedFiles`: esso associa i dati (ovvero il body) di un file recuperato dalla rete con la chiave "`<host>/<pathFile>`" recuperata dalla request. Il metodo `handle()` recupera (dalla request) i parametri che formano la chiave e verifica se esiste una corrispondenza nella map; se non esiste viene istanziata una socket e si tenta di aprire una connessione nel seguente modo:

```
Socket socket = new Socket(request.getHost(), 80);
```

Se la connessione ha avuto successo (ovvero se non è stata sollevata nessuna eccezione del tipo `UnknownHostException`), allora il metodo privato `redirectRequest()` inoltra la request ricevuta dal proxy all'host destinatario, infine si legge la risposta nel metodo privato `getReply()`. Il metodo `handle()` restituisce la reply se è stata correttamente creata, altrimenti null.

Un oggetto `HTTPProxyHandler` dovrebbe essere inserito come primo handler dal server (per approfondire leggere paragrafo 2.4 `HandlersPool`), in modo da intercettare tutte le richieste pervenute.

Nell'esempio seguente è stata richiesta la pagina `index.html` del sito "`www.google.it`" per 2 volte, la quale viene stampata su standard output dello script python (a sinistra). Notare che non viene salvata l'intestazione originale, ma solo il corpo. La seconda volta la pagina viene recuperata dal proxy.

```
iMac-di-Matteo:src matteomauro$ python3 reqProxy.py
----- First request -----
HTTP/1.1 200 OK
X-Frame-Options: SAMEORIGIN
Accept-Ranges: none
Cache-Control: private,
Server: gws
Connection: close
Set-Cookie: NID=122=eZnONxLDvhLRnEv8DScwVr-s7pWaYRZRph
uoljdesgvb3Uyx0BSg1JOKJIAjndFo_iyC8sCRe8BdDXNCH7vpQXv1
VhSrnxRxsGsbPuFFUQNgW8kKnNMScbEVMN503zk;
Vary: Accept-Encoding
Expires: -1
P3P: CP="This
X-XSS-Protection: 1;
Date: Sun,
Content-Type: text/html;

----- Second request -----
HTTP/1.1 200 OK
Content-Length: 0
Date: dom, 21 gen 2018 12:15:38 CET
```

```
Server listening...
Request accepted.

Server listening...
Request accepted.

Server listening...
The file has been successfully retrieved from the proxy.
```

2.2 Streams

2.2.1 Classe: MyHTTPInputStream

La classe MyHTTPInputStream possiede un campo privato di tipo BufferedReader per eseguire la lettura (carattere per carattere) dallo stream ricevuto da parte della socket. Ho deciso di optare per un BufferedReader (nonostante la lettura venga eseguita attraverso la read() e non readLine()) in quanto risulta più efficiente rispetto all'InputStreamReader, dal momento che il contenuto viene preventivamente letto e bufferizzato (come riportato qui: <https://www.quora.com/What-is-the-difference-between-BufferedReader-and-InputStreamReader-in-Java>).

La classe delega alla classe HTTPMessageParser il compito di recuperare e creare una HTTPRequest dallo stream.

2.2.2 Classe: MyHTTPOutputStream

Questa classe scrive, attraverso un oggetto di tipo OutputStream, la request/reply desiderata convertendo il messaggio da inviare in byte secondo la codifica ASCII a 7 bit, rappresentata da java dall'attributo statico "US_ASCII" della classe StandardCharsets. (come evidenziato dalla documentazione: https://docs.oracle.com/javase/7/docs/api/java/nio/charset/StandardCharsets.html#US_ASCII).

2.2.3 Classe: HTTPMessageParser

La classe più interessante è rappresentata da HTTPMessageParser, in quanto esegue l'effettivo parsing dei dati letti e ne costruisce una HTTPRequest/HTTPReply se il contenuto risulta semanticamente corretto, altrimenti lancia un'eccezione HTTPProtocolException evidenziando l'errore specifico.

Partendo dal presupposto che la lettura dallo stream è un'operazione analoga sia per una request che per una reply, ho deciso di generalizzare la classe per questo compito facendo decidere all'utente quale operazione eseguire: la classe offre il costruttore:

```
public HTTPMessageParser(BufferedReader reader, Type type)
throws IllegalArgumentException
```

la classe `Type` è una classe interna ed è rappresentata da un'enumerazione:

```
public static enum Type {  
    REQUEST, REPLY  
};
```

in tal modo un `HTTPMessageParser` può essere istanziato per gestire una request oppure una reply, senza aver dovuto realizzare due classi separate (in tal modo è possibile sfruttare i metodi privati comuni). Se viene passato un type non corretto, viene lanciata un'eccezione. Ovviamente un `HTTPMessageParser` creato per gestire una request non potrà mai gestire una reply e viceversa.

Esempio:

```
HTTPMessageParser parser = new HTTPMessageParser(reader, Type.REQUEST);  
/*  
    parser è abilitato ad eseguire una parseRequest(), ma non parseReply().  
    Nessun valore diverso da quelli enumerati in Type può essere accettato.  
*/
```

I due metodi pubblici che la classe `HTTPMessageParser` rende disponibili sono:

- `public HTTPRequest parseRequest() throws HTTPProtocolException, UnsupportedOperationException`
- `public HTTPReply parseReply() throws HTTPProtocolException, UnsupportedOperationException`

L' `UnsupportedOperationException` è necessaria nel caso che un `HTTPMessageParser` creato per gestire un determinato type sia richiamato attraverso il metodo sbagliato.

Ogni request/reply prevede 3 sezioni ben separate:

- request line (`HTTPRequest`) / status line (`HTTPReply`) -> esattamente una riga;
- header section (`HTTPRequest/HTTPReply`) -> 1 o più righe;
- body ((`HTTPRequest/HTTPReply`) -> numero di byte precisato da `Content-Length`.

Pertanto il metodo privato `readLine()` si occupa di leggere una riga per volta il contenuto dello stream, dove ogni riga è rappresentata dal raggiungimento del carattere `'\n'`.

Prendendo come esempio il metodo `parseRequest`, i passi principali sono:

- 1) richiama `parseRequestLine()` per identificare la request line;
- 2) se la request line è diversa da null, richiama `parseHeaderLines()` per identificare i parametri;
- 3) chiama il metodo `readBody(long length)` per leggere tanti byte quanti specificati da `Content-Length` (se non era presente allora di default è settato a 0).

L'eccezione `HTTPProtocolException` è lanciata qualora la struttura di una delle righe non sia conforme allo standard: per fare ciò ho sfruttato delle regular expression costruite nel seguente modo:

Legenda:

- `"."` -> almeno un carattere alfanumerico
- `"[01]"` -> 0 oppure 1
- `"\d+"` -> almeno un carattere numerico
- `"[\w\s]+"` -> uno o più tra lettere e spazi
- `"^"` -> inizio linea (ovvero spazi iniziali non ammessi)
- `"$"` -> carattere terminatore

pattern per riconoscere una request line semanticamente corretta:

-> Pattern `patternFirstLine = "^.+ HTTP/1.[01]$"`
ovvero rappresenta `"<method> <path> HTTP/<version>"`

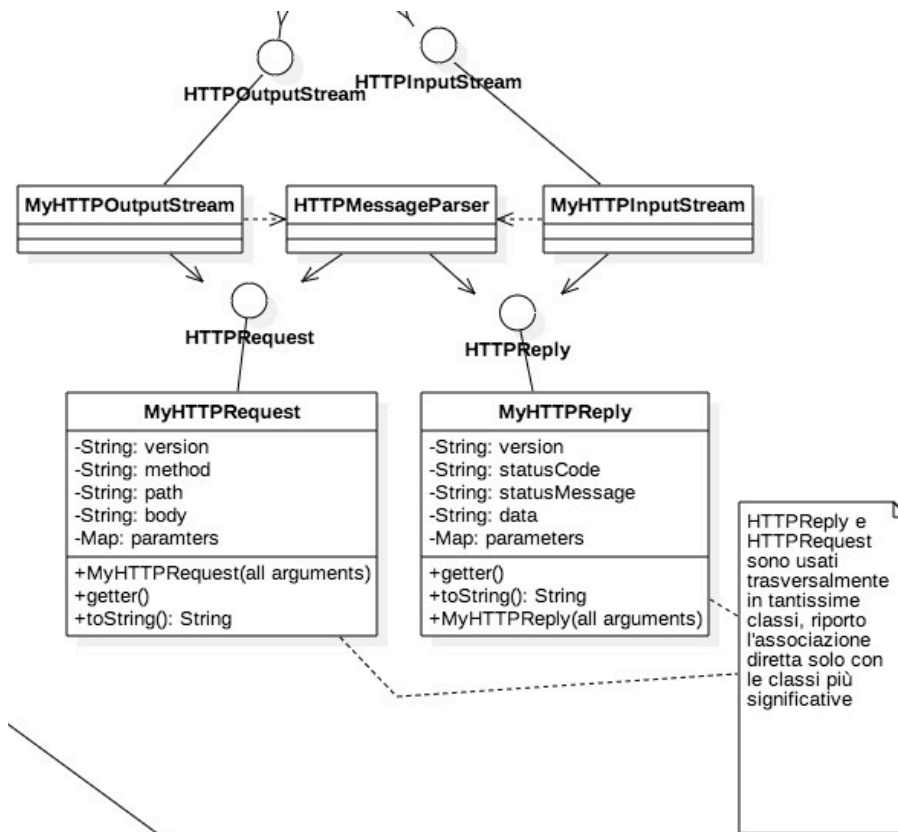
pattern per riconoscere una status line semanticamente corretta:

-> Pattern `patternFirstLine = "^HTTP/1.[01] \d+ [\w\s]+ $"`

pattern per riconoscere una header line semanticamente corretta:

-> Pattern `patternHeaderLine = "^.:.+$"`
ovvero rappresenta `"<parameter>: <value>"` (notare lo spazio dopo i due punti).

Attenzione: come scelta implementativa, una request line non corretta non consente di proseguire nel costruire una request (ritorna null), mentre una header line non corretta viene semplicemente scartata ma non porta necessariamente ad una request non valida (discorso analogo per una reply). Le regular expression riconoscono solo se la struttura è semanticamente giusta, non controlla se il metodo descritto o altri valori sono logicamente giusti (tali controlli sono eventualmente eseguiti successivamente dagli handlers).



Struttura UML degli streams e di HTTPMessageParser

2.3 Classe: MyHTTPServer

Il costruttore di questa classe istanzia una SocketServer, che verrà effettivamente lanciata all'ascolto con il metodo start(). Per ogni richiesta di connessione ricevuta viene creato e fatto partire un oggetto di tipo HTTPReplyThread, ovvero un thread capace di gestire la request.

MyHTTPServer contiene un attributo di tipo HandlersPool, ovvero una classe implementata come un monitor che viene condiviso (viene passato al costruttore) di ogni HTTPReplyThread: un oggetto di tipo HandlersPool rappresenta un insieme di oggetti che implementano l'interfaccia HTTPHandler, che possono essere dinamicamente aggiunti al gruppo (per una descrizione più esauriente guardare paragrafo 2.4).

Il metodo stop() del server comporta la sola chiusura della socketServer, ma non avrà ripercussioni su eventuali connessioni precedentemente aperte da altri clients, che saranno dunque liberi di proseguire e terminare le proprie sessioni di invio/ ricezione dati.

2.4 Classe: HandlersPool

Se per ipotesi il server ricevesse 10 richieste quasi contemporaneamente e ogni thread istanziasse indipendentemente 8 handlers a testa (GET x2, HEAD x2, POST x2, PUT x1, DELETE x1), ci sarebbero 80 oggetti diversi in memoria (la maggior parte dei quali potrebbero non essere nemmeno chiamata durante la ricerca dell'handler compatibile con la richiesta), ognuno dei quali sarebbe immediatamente distrutto con la terminazione del thread proprietario. Per evitare tale proliferazione di oggetti ho realizzato la classe HandlersPool, la quale viene istanziata dal server: il metodo addHandler() di quest'ultimo permette di aggiungere al pool un numero arbitrario di HTTPHandler (ovvero tutte le classi descritte nel paragrafo 2.1); ovviamente tale compromesso impedisce ai thread di lavorare effettivamente in parallelo, ma risulta più efficiente nell'utilizzo di risorse di memoria.

Il metodo synchronized handle() itera la lista di handlers correnti ed è coerente con il comportamento dei "veri" HTTPHandler: restituisce una HTTPReply se la request è stata correttamente gestita da uno di essi, altrimenti ritorna null.

Questa classe è l'unica ad avere accesso sia alla request ricevuta che all'effettiva reply generata, pertanto la funzionalità di log del server è implementata in questa classe, nel metodo privato log(): il file "log.txt" (aperto attraverso getClass().getResource() che coincide con il file presente nella cartella bin/log/) viene aggiornato da un ulteriore Thread, definito con una lambda expression e lanciato separatamente per impedire di arrecare troppo ritardo alla scrittura della reply nei confronti del client.

La struttura del file di log è la seguente:

```
- - - - -  
REQUEST - <data e ora>  
<testo request>  
    .  
    .  
    .  
<fine request>  
  
REPLY - <data e ora>  
<testo reply>  
    .  
    .  
    .  
<fine reply>  
- - - - -
```

2.5 Classe: HTTPReplyThread

Ogni thread ha un riferimento all'HandlersPool ricevuto con il costruttore, un riferimento alla socket connessa con il client e i due stream di input e output recuperati dalla socket stessa, dei quali il thread è ovviamente responsabile dell'apertura e chiusura attraverso i metodi privati `instantiateStreams()` e `closeStreams()`.

Il metodo `run()` delega al pool la gestione della request recuperata dall'`HTTPInputStream`, infine ottiene una `HTTPReply` che invia al client attraverso l'`HTTPOutputStream`: in tutto questo il thread deve verificare se chiudere la connessione o meno dopo l'invio, quindi per far ciò si cerca all'interno della request la riga "Connection":

- se presenta "keep-alive", allora gli stream e la socket rimangono aperti (in particolare la lettura sullo stream di input è bloccante, rimanendo in attesa di ulteriori dati);
- se presenta "close", allora viene richiamato il metodo `closeStreams()` e il thread termina l'esecuzione.

Ovviamente la linea "Connection" potrebbe non essere presente nella request, in tal caso di default una `HTTPRequest 1.0` è non persistente, mentre una `HTTPRequest 1.1` è persistente.

Infine se nessun handler è stato in grado di gestire la request, allora viene inviato un messaggio “501 Not Implemented Method”, oppure se la request era sintatticamente non corretta viene inviato un “400 Bad Request” (rilevata da una `HTTPProtocolException`).

2.6 Classe: `MyHTTPRequest` / `MyHTTPReply`

`MyHTTPRequest` contiene 5 campi privati, rispettivamente:

- 1) `String version` (HTTP/1.0 oppure HTTP/1.1);
- 2) `String method` (GET, POST ecc);
- 3) `String path` (URL del file recuperato dalla request line);
- 4) `String body`;
- 5) `Map<String, String> parameters` (rappresentazione delle coppie chiave-valore della header section).

La classe offre solo metodi getter, il metodo `toString()` fornisce una rappresentazione ben formattata della request.

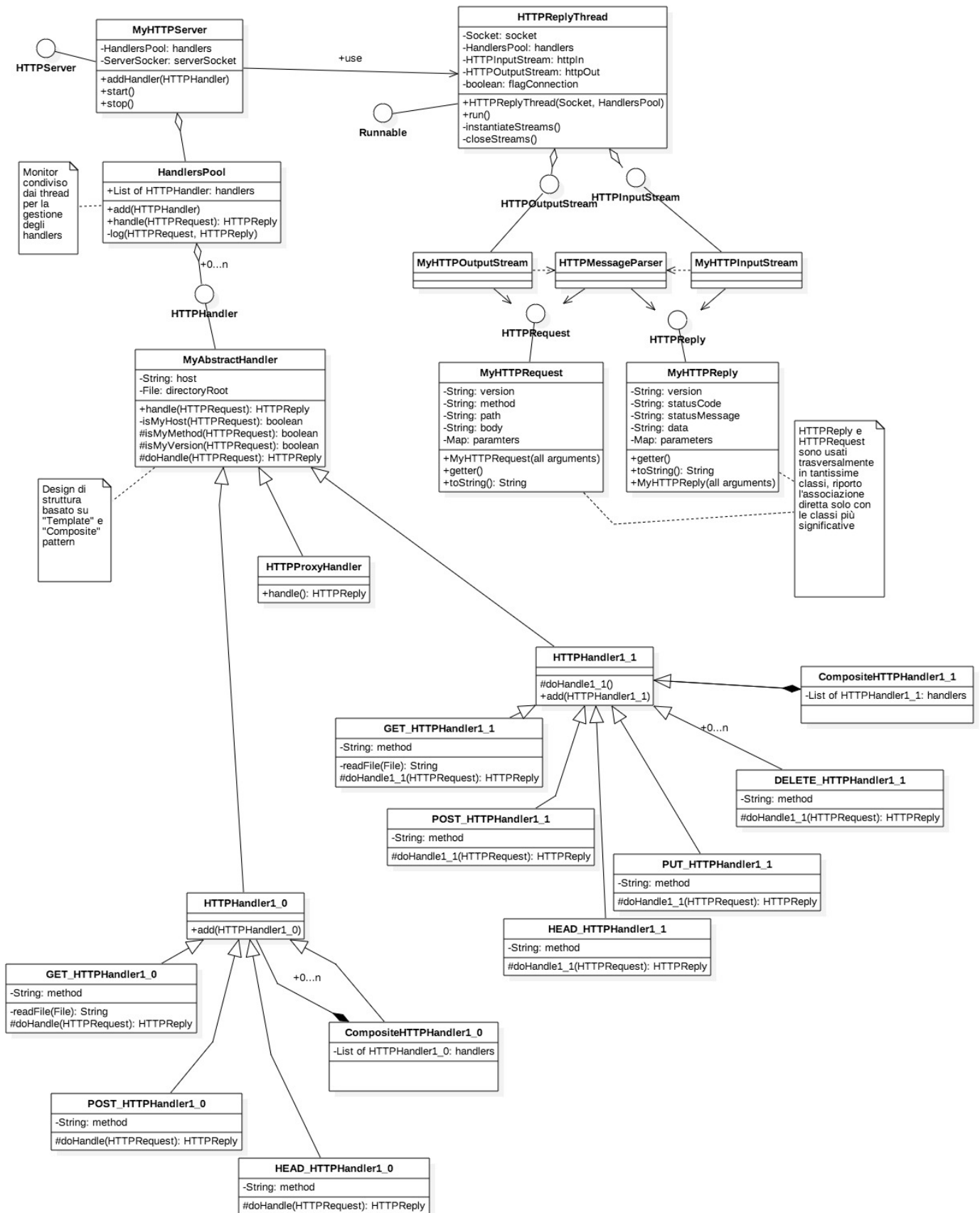
`MyHTTPReply` contiene 5 campi privati, rispettivamente:

- 1) `String version` (HTTP/1.0 oppure HTTP/1.1);
- 2) `String statusCode` (200, 404 ecc);
- 3) `String statusMessage` (“OK”, “File Not Found” ecc);
- 4) `String data`;
- 5) `Map<String, String> parameters` (rappresentazione delle coppie chiave-valore della header section).

La classe offre solo metodi getter, il metodo `toString()` fornisce una rappresentazione ben formattata della reply.

3.0 Schema UML

3.0 Schema UML

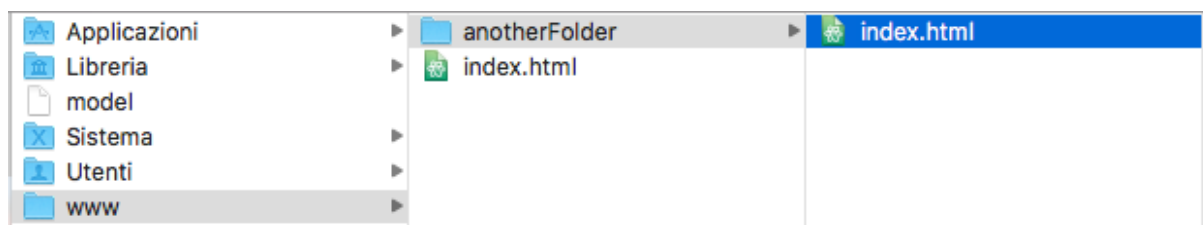


4.0 Test

Premessa per il corretto funzionamento dei test: le classi JUnit Test implementate sono pensate per funzionare utilizzando le risorse nelle cartelle "testSamples/" e "testSamples/methodSamples/". I file .txt presenti contengono richieste e risposte HTTP "prefatte" per essere usate nei test.

Inoltre è fondamentale inserire nella root del file system del proprio PC le cartelle www/ e www/anotherFolder/ con all'interno i file index.html, tali risorse saranno necessari alla simulazione di funzionamento degli handler (una copia di queste risorse è fornita insieme al progetto, in particolare il file index.html deve contenere solo la scritta "SAY PEACE!", esattamente 10 caratteri). Tali cartelle saranno fornite insieme al progetto di consegna per essere eventualmente copiate direttamente.

File System Root contenente "www" -> cartella anotherFolder + index.html -> index.html



4.1 [GET/HEAD/POST]_HTTPHandler[1.0/1.1]Test

Le classi Test per gli handler eseguono tutte gli stessi test, con le ovvie differenze di implementazione.

@Test

```
public void validRequest():
```

verifica che una reply sia correttamente generata nei seguenti 3 casi:

- host e root non specificati per l'handler (default -> host = null, root="/www");
- solo l'host specificato;
- solo la directoryRoot specificata.

@Test

```
public void fileNotFound404():
```

verifica che un "404 File Not Found" vengano generato se il file non è presente.

```
@Test
```

```
public void hostNotCompatible():
```

verifica che la reply sia negata nel caso che la request non contenga l'host corretto.

```
@Test
```

```
public void methodNotCompatible():
```

verifica che la reply sia negata nel caso che la request non contenga il metodo compatibile.

4.2 HTTPMessageParserTest

```
@Test(expected = IllegalArgumentException.class)
```

```
public void illegalArgumentExceptionHTTPMessageParser():
```

verifica che lanci l'eccezione IllegalArgumentException se si passa un argomento non valido per il Type.

```
@Test
```

```
public void parseRequest_Valid():
```

controlla che una request venga correttamente estrapolata dal buffer e che il parsing venga efficacemente eseguito, verificando che l'oggetto HTTPRequest recuperato contenga i giusti campi.

```
@Test(expected = UnsupportedOperationException.class)
```

```
public void parseRequest_WrongCall() throws
```

```
HTTPProtocolException, UnsupportedOperationException:
```

verifica che lanci la corretta eccezione se un oggetto HTTPMessageParser viene chiamato per fare il parsing di una reply sebbene sia stato creato specificatamente per le request.

```
@Test(expected = HTTPProtocolException.class)
```

```
public void parseRequest_WrongRequestLine() throws
```

```
HTTPProtocolException, UnsupportedOperationException:
```

verifica che una request line non corretta all'interno di una HTTPRequest sia correttamente rilevata, lanciando l'eccezione HTTPProtocolException.

```
@Test
```

```
public void parseRequest_WrongHeader():
```

controlla che una header line non corretta sia rilevata con successo (nell'esempio fornito nel test nessuna header line è corretta, quindi vengono scartate tutte ed il conteggio finale deve risultare 0).

```
@Test
```

```
public void parseReply_Valid():
```

controlla che una reply venga correttamente estrapolata dal buffer e che il parsing venga efficacemente eseguito, verificando che l'oggetto HTTPReply recuperato contenga i giusti campi.

```
@Test(expected = HTTPProtocolException.class)
```

```
public void parseReply_WrongStatusLine() throws
```

```
HTTPProtocolException, UnsupportedOperationException:
```

verifica che una status line non corretta all'interno di una HTTPReply sia correttamente rilevata, lanciando l'eccezione HTTPProtocolException.

4.3 MyHTTPInputStreamTest

```
@Test
```

```
public void readHTTPRequest_Valid():
```

verifica che i dati nello stream (che rappresentano una HTTPRequest) siano correttamente letti ed interpretati.

```
@Test(expected = HTTPProtocolException.class)
```

```
public void readHTTPRequest_SemanticWrong()
```

```
throws HTTPProtocolException, UnsupportedOperationException:
```

verifica che la lettura di una HTTPRequest non sintatticamente corretta ritorni null (gli errori sintattici sollevano un'eccezione).

```
@Test
```

```
public void readHTTPReply_Valid():
```

verifica che i dati nello stream (che rappresentano una HTTPReply) siano correttamente letti ed interpretati.

```
@Test(expected = HTTPProtocolException.class)
```

```
public void readHTTPReply_SemanticWrong()
```

```
throws HTTPProtocolException, UnsupportedOperationException:
```

verifica che la lettura di una HTTPReply non sintatticamente corretta ritorni null (gli errori sintattici sollevano un'eccezione).

5.0 Guida all'utilizzo del programma e degli script Python

Istanziare un oggetto di tipo HTTPServer dalla classe MyHTTPFactory, attraverso il metodo getHTTPServer(). E' possibile aggiungere gli handlers desiderati sia direttamente con il costruttore che attraverso il metodo addHandler().

Attenzione: gli handler saranno richiamati secondo un ordine FIFO, ovvero nell'esatta sequenza con cui sono stati inseriti, pertanto se si è interessati ad utilizzare la classe HTTPProxyHandler si consiglia di aggiungerla per prima.

Gli handler appartenenti alla stessa versione di protocollo sono raggruppabili in composti come nel seguente esempio:

```
CompositeHTTPHandler1_0 handler = new CompositeHTTPHandler1_0();
try {
    handler.add(new GET_HTTPHandler1_0(root));
    handler.add(new POST_HTTPHandler1_0(root));
    handler.add(new HEAD_HTTPHandler1_0(root));
} catch (Exception e) {
    e.printStackTrace();
}
```

Dopo aver istanziato il server, lanciare il metodo start() per iniziare la fase di ascolto delle richieste dei clients. Il server mostra questo messaggio:

Server listening...

Per ogni request ricevuta, viene mostrato il seguente messaggio e ritorna all'ascolto:

Server listening...
Request accepted.
Server listening...

Se la richiesta ricevuta viene direttamente recuperata dal server proxy, viene mostrato questo messaggio:

The file has been successfully retrieved from the proxy.

ATTENZIONE: utilizzare Python3 per eseguire gli script, print è utilizzato come funzione “print(<something>)” e non come statement.

Gli script Python forniti sono i seguenti:

- reqNonPersistent.py: contiene 8 funzioni, ognuna delle quali rappresenta l’invio di una richiesta NON persistente per ogni metodo implementato, l’indirizzo della socket server destinatario è “localhost:<port>” (<port> deve essere passata come argomento allo script, vedere l’esempio riportato in fondo); ogni reply ricevuta viene stampata su standard output;
- reqPersistent.py: lo script invia 4 richieste al server, le prime 3 (tutte di tipo GET per semplicità) contengono il parametro “Connection: keep-alive” e vengono quindi inviate in pipelining sulla stessa connessione, infine il quarto e ultimo messaggio contiene “Connection: close” per terminare la connessione.
- reqProxy.py: invia 2 richieste GET con lo scopo di recuperare da Internet la pagina “index.html” della pagina “www.google.it”; i messaggi vengono inviati all’handler proxy del server: mentre la prima sarà effettivamente inoltrata a “www.google.it:80”, la seconda richiesta sarà direttamente soddisfatta dal proxy stesso.

L’indirizzo IP con il quale gli script dialogano deve essere sempre “localhost” quindi è stato implementato “hard-coded”, mentre per il numero di porta ho deciso di lasciare libero l’utente di specificarlo come argomento dello script (ovviamente deve combaciare con quello fornito al server Java, nell’esempio seguente è 6000).

Esempio corretto

```
iMac-di-Matteo:src matteomauro$ python3 reqNonPersistent.py 6000
----- GET 1.0 -----
HTTP/1.0 200 OK
Content-Length: 10
Date: gio, 25 gen 2018 11:29:35 CET

SAY PEACE!
```

Controlli sui valori inseriti:

1) nessun argomento

```
iMac-di-Matteo:src matteomauro$ python3 reqNonPersistent.py
----- GET 1.0 -----
ERROR: you must define a port number.
```

2) argomento non numerico

```
iMac-di-Matteo:src matteomauro$ python3 reqNonPersistent.py qwe
----- GET 1.0 -----
ERROR: port is not alphanumeric.
```