Laboratorio di Basi di Dati e Programmazione Web

Programmazione web
Cenni di HTTP ed il protocollo CGI

Stefano Zacchiroli
zack@pps.jussieu.fr

Programmazione Web

Cenni di HTTP

HTTP

- HTTP è un protocollo di “VII livello”
  - Stateless
  - Client-server
  - Request-response
  - Nato con il primo browser, standardizzato successivamente da IETF, ora mantenuto da W3C
  - Text-based
- Reference: http://tools.ietf.org/html/rfc2616

- Caratteristiche
  - Content negotiation
  - Caching, multi-tier pipeline architecture
  - autenticazione
- Ruoli
  - **Client**: un'applicazione che stabilisce una connessione HTTP. Scopo: mandare richieste
  - **Server**: un'applicazione che attende e accetta connessioni, per generare risposte.
  - Una connessione / molte richieste-risposte

Richiesta

- La richiesta più semplice:

```
GET URI CrLf
```

- Sinossi generale:

```
Method URI Version CrLf
[Header*]
CrLf
[Body]
CrLf
```

- Reference
  http://tools.ietf.org/html/rfc26 16#section-5

- *Method :* azione richiesta dal client
- *URI* : identificativo di risorsa relativo al server
- *Header* : linee RFC822
  - i.e. mail-like header
- *Body* : messaggio MIME
- *Version*
  - *“HTTP/1.0”* o *“HTTP/1.1”*
  - Se presente, il server attende per Header+Body, altrimenti risponde immediatamente

Richiesta – esempio

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (X11; U; Linux x86_64; en-US;
rv:1.9.0.10) Gecko/2009042805 Firefox/3.0.1
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,
*/*;q=0.8
Accept-Language: en-us,en;q=0.7,it;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: … <snip> …
```

- Consiglio: estensione “Live HTTP Headers” di Firefox

Richiesta – metodi

- I metodi più comunemente usati di HTTP
  - GET: richiede (la rappresentazione di) una risorsa
  - HEAD: richiede gli header di una risorsa
  - POST: invia contenuto ad una risorsa destinatario
- Più raramente (soprattutto in WebDAV)
  - PUT: crea una nuova risorsa
  - DELETE: rimuove una risorsa
- Metodi idempotenti: GET, HEAD, PUT, DELETE
- Metodi sicuri (non cambiano lo stato del server): GET, HEAD
  - Nota: POST non è ne sicuro ne idempotente, va usato come tale
- Reference: http://tools.ietf.org/html/rfc2616#section-9

Header “notevoli”
(generali e di entità)

- Date: timestamp della connessione
- Connection: tipo di connessione (persistente o meno?)
- Content-Type: MIME type (obbligatorio se c'è BODY)
- Content-Length: lunghezza del BODY
  - Nota: è indispensabile per capire dove termina il BODY, in caso di connessioni persistenti
- Content-Encoding
- Expires: cache control timestamp
- Last-Modified: timestamp sull'ultima modifica
  - Usato tipicamente in congiunzione con il metodo HEAD per capire se è necessario ri-scaricare una risorsa o meno
- Reference: http://tools.ietf.org/html/rfc2616#section-4.2

Richiesta – header “notevoli”

- User-Agent
- Referer (typo voluto)
- Host
  - Obbligatorio in HTTP/1.1, identifica il nome di virtual host per il quale si richieda l'accesso a risorse
- Accept-{Charset,Encoding,Language}: content negotiation
- Reference: http://tools.ietf.org/html/rfc2616#section-5.3

Risposta

- Sinossi generale:

Version Code Reason CrLf

[*Header**]

CrLf

[*Body*]

CrLf

- Reference: http://tools.ietf.org/html/rfc2616#section-6

- Version: come per la richiesta (in sync)
- Status code: tipo di risposta
  - 1xx: informational
  - 2xx: success
  - 3xx: redirection
  - 4xx: client error
  - 5xx: server error
- Reason: “explanation”
- Body: messaggio MIME (con header)

Risposta – esempio

```
HTTP/1.1 200 OK
Cache-Control: private, max-age=0
Date: Sun, 03 May 2009 08:59:03 GMT
Expires: -1
Content-Type: text/html; charset=UTF-8
Set-Cookie: GTZ=; expires=Mon, 01-Jan-1990
00:00:00 GMT; path=/; domain=.google.com
Content-Encoding: gzip
Server: gws
Content-Length: 3357
```

Status code “notevoli”

- 100 Continue
- 200 Ok
- 201 Created
- 301 Moved permanently
- 400 Bad request
- 401 Unauthorized
- 403 Forbidden
- 404 Not found
- 500 Internal server error
- 501 Not implemented
- Reference: http://tools.ietf.org/html/rfc2616#section-10

Risposta – header “notevoli”

- Sono permessi gli header generali e di entità già visti
- Server: “signature” del server
- Www-Authenticate: usato per richiedere autenticazione (vedi dopo)
- Reference: http://tools.ietf.org/html/rfc2616#section-6.2

Esercizi

- Implementare in Python un semplice client HTTP, che implementi HTTP/1.0 (senza usare librerie!)
  - Dato un URL a command line, scarica la risorsa corrispondente (usando il metodo GET di HTTP) e la salva su un file specificato a command line
  - Opzionalmente, non scarica la risorsa ma ne mostra solo gli header (usando il metodo HEAD di HTTP)
- Implementare in Python un semplice server HTTP, che supporti un piccolo frammento di HTTP/1.1 (… senza usare liberie!)
  - Pagine HTML statiche
  - Directory listing della singola directory ove è in esecuzione
  - Testarlo con un browser a vostra scelta

Autenticazione – “basic” scheme

1. Client invia richiesta
2. Riceve risposta 401 (unauthorized) con header:
   - `WWW-Authenticate: Basic realm="WallyWorld"`
3. (Il client chiede auth info all'utente)
4. Il client re-invia la richiesta precedente con header
   - `Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==`
   - Dove “Basic” è seguito dall'encoding Base64 di *USERNAME “:” PASSWORD*
   - Nota: username e password viaggiano quindi in chiaro
5. (Il client solitamente fa cache di auth info e le riusa per autenticazione futura dello stesso realm)

- Reference: http://tools.ietf.org/html/rfc2617

Esercizi

- Aggiungere il supporto per autenticazione basic al server HTTP sviluppato in precedenza
  - La richiesta di un path a scelta (e.g. /secret) deve richiedere autenticazione e permettere accesso solo ad un utente con password a scelta (e.g.: utente “pippo”, password “foobar”)

Programmazione Web

Il protocollo CGI

CGI – motivazioni

- HTTP è nato come protocollo di scambio di risorse (scientifiche): inizialmente read-write, poi de facto read-only
- Successivamente è diventato il protoccolo più “accessibile” per tutti gli utenti
  - Metafora: “browser come sistemi operativi”
- Nuova necessità: usare HTTP come portale per accedere ad applicazioni le più disparate
- Soluzione
  1. CGI: protocollo per esporre su Web applicazioni
  2. AJAX: stack tecnologico per aumentare l'interattività
     - Migliora la user-experience

CGI – Common Gateway Interface

- CGI è uno *protocollo* per interfacciare applicazioni esterne con information server
  - Applicazioni: le più disparate, eseguibili in qualche OS
  - Information server tipico: server HTTP
- Di fatto, CGI introduce un nuovo tipo di risorsa
  - Finora: un server HTTP serve risorse statiche
  - Con CGI: un server HTTP serve risorse dinamiche generate da un'applicazione esterna. Il server HTTP agisce da *tramite* (gateway) tra lo user agent è l'applicazione esterna
- Lo standard CGI stabilisce come l'applicazione esterna (*CGI Program* d'ora in poi) comunichi con l'information server
- Reference; http://hoohoo.ncsa.illinois.edu/cgi/interface.html
  - Versione 1.2 “in progress” da sempre: http://www.w3.org/CGI/

CGI program

- Un *CGI Program* è un eseguibile all'interno di un OS
  - Può essere implementato in *qualsiasi* linguaggio di programmazione
    - Tipicamente, ma solo per comodità di deployment, si tratta di linguaggi interpretati ...
    - … ma un CGI può anche essere implementato in C
  - mod_{perl,python,tcl,...} non costituiscono programmi CGI
- Problemi di sicurezza
  - CGI permette di eseguire programmi attraverso il Web
  - Precauzioni tipiche: una dir specifica, controlli di accesso, exec mode, SUID
  - … tema sterminato, non ce ne occuperemo in dettaglio
    - cfr.: corso di sicurezza

Il protocollo CGI

- Il server HTTP è il principale attore di CGI:
  - Identifica URL corrispondenti a CGI program
  - Esegue l'eseguibile corrispondente
    - Senza cmdline args
    - Con environment vars stabilite dallo standard CGI
  - Restituisce lo standard output del programma al client
    - L'output costituisce BODY + HEADER
    - Gli HEADER sono complementati da alcuni header standard forniti dal server (e.g. “Date:”)
- Comunicazione
  - Client → CGI program: avviene via variabili d'ambiente
  - CGI program → client: avviene via standard output
  - In entrambi i casi, il server HTTP funge da mediatore

CGI – variabili d'ambiente

- Sia la richiesta CGI corrispondente all'URL:

  http://www.example.com/path/in/fo/foo.cgi?query

- Variabili d'ambiente passate a foo.cgi per ricevere argomenti dall'invocazione web:
  - QUERY_STRING: qualsiasi cosa segua '?' nell'URL
    - Origine usuale: form HTML che usi metodo GET
    - Semantica: information query, i.e. cosa lo user agent “chiede” al CGI program
    - URL encoding: “+” al posto di spazi, %XX per caratteri specificati in esadecimale
  - PATH_INFO: path che segue il server name, fino a '?', esso escluso
    - Semantica: percorso di accesso al CGI program

HTML forms

```
<form method="GET" action="/path/foo.cgi">
  Name: <input type="text" name="name" /><br />
  Surname: <input type="text" name="surname" /><br />
  Email: <input type="text" name="email" /><br />
  Passsword: <input type="password" name="pwd" /><br />
  Subscribe: <input type="checkbox" name="sub" value="sub"/><br />
  Gender: <input type="radio" name="sex" value="male" /> Male
    <input type="radio" name="sex" value="female" /> Female <br />
  Comment: <input type="textarea" name="comment" /><br />
  How did you find us: <select name="feedback">
    <option value="default">I did not!</option>
    <option value="friends">Friends</option>
    <option value="web">Web search</option>
  </select>
  <input type="submit" value="submit" />
  <input type="reset" />
</form>
```

- Sample request risultante:

http://localhost/path/foo.cgi?name=Stefano&surname=Zacchiroli&email=zack%40pps.jussieu.fr&pwd=secret&sub=sub&sex=male&comment=thanks+a+lot!&feedback=default

GET & URL encoding

- Come visto nell'esempio precedente, usando i form con il metodo GET, tutti i dati forniti dall'utente sono passati dallo user agent nell'URL della richiesta
- Le regole di codifica (di default) prendono il nome di “WWW Form URL encoding” (application/x-www-form-urlencoded)
  - Ad ogni campo <input> corrisponde un nome, specificato dall'attributo name; il suo valore dipende dalle scelte dell'utente
  - Ogni nome viene concatenato al suo valore usando “=” come separatore
  - I frammenti risultanti vengono concatenati usando “&” come separatore
  - La stringa risultante viene URL-encoded (“+” e “%XX”)
  - Si ottiene così la query string, che viene concatenata al URL del CGI usando “?” come separatore
- Per ottenere i singoli parametri occorre effettuare la procedura inversa

Form e metodo POST

- `<form method=”get”>`
  - Richiesta HTTP con metodo “GET”
  - Dati passati via QUERY_STRING
  - Size limitation, browser-dependent
  - Sensitive information visbili nella barra del browser
- `<form method=”post”>`
  - Richiesta HTTP con metodo “POST”
  - Dati passati via request BODY: accessibile dal CGI su standard input, bisogna tenere conto della lunghezza del body
  - No size limitation
  - Non bookmark-safe

CGI – specification highlights

- Variabili d'ambiente
  - SERVER_NAME: nome del server HTTP, URL “base”
  - SERVER_PORT: porta TCP del server
  - REQUEST_METHOD: metodo HTTP
  - PATH_INFO, QUERY_STRING
  - REMOTE_HOST, REMOTE_ADDR, ...: client info
  - CONTENT_TYPE, CONTENT_LENGTH: body info
    - Solo per richieste con BODY
- Standard input
  - Per richieste con BODY, esso viene passato via STDIN
- Standard output
  - Usato per passare al client una risorsa generata dallo script, o informazioni di redirezione

CGI – generare output

- Un programma CGI *non* deve generare risposte HTTP nella loro interezza
  - lo scheletro e gli header non dipendenti dalla risposta vengono generati dal server
- Un programma CGI può generare
  - Header HTTP stampandoli nella forma “Name: Value” CRLF
    - Deve sempre generare l'header “Content-type”, solitamente “Content-type: text/html”
    - Al termine degli header deve sempre generare una linea vuota
  - BODY: tutto ciò che segue la linea vuota e si estende fino alla chiusura di STDIN costituisce il BODY della risposta
  - *Server directive* per indicazioni al server il tipo di risposta
    - “Status: xxx”: per modificare lo status code
    - “Location: URL” per richiedere un redirect

Hello, world!(s)

```
#!/usr/bin/python
print “Content-type: text/html”
print
print “””<html><body>
  Hello, world!
</body></html>”””
```

```
#!/usr/bin/python
print “Location: “,
print “http://www.google.com”
print
```

```
#!/usr/bin/python
print “Content-type: text/html”
print “Status: 404”
print
print “””<html><body>
  <h1>Not found</h1>
  <p>The document you requested
  was not found on this server
  </p>
  </html></body>”””
```

Info – usare CGI in lab

- Ogni utente disponde di una dir `/home/web/USERNAME/` con layout
  - `cgi-bin/` → file cgi abilitati per essere eseguiti
    - I file python devono comunque avere estensione .cgi ed essere eseguibili
  - `data/` → file per i dati
    - I .cgi possono referenziarli via ../data o path assoluto
  - `html/` → file HTML statici, è la DocumentRoot
  - `log/` → log file di apache
    - access.log: contiene tutti gli accessi
    - error.log: contiene gli errori, inclusi i backtrace di .cgi Python

Esercizi

- Implementate con form HTML + Python CGI un front-end ad alcune utility di networking disponibili su *NIX
  - finger
  - dhois
  - dig / host
- Implementate in Python un CGI che permetta directory browsing: non solo la dir dove risiede, ma anche sub- e parent- directory
- Implementare in Python il CGI man.cgi che restituisce in formato HTML una manpage specificata come parametro

Esercizi

- Implementate un semplice wiki come CGI Python
  - Ad ogni pagina corrisponde un file .txt su disco, con sintassi a vostra scelta
  - Un CGI view.cgi effetua il rendering di un file.txt specificato con path info
    - Hint: usate il modulo di rendering per sintassi wiki sviluppato in precedenza
  - Un CGI edit.cgi permette all'utente di modificare la pagina ritornando un form con il contenuto testuale
  - Un CGI save.cgi salva una pagina ed effettua una redirezione verso view.cgi
  - *Bonus*: Estendete il wiki in modo che tutti i file .txt al di sotto di una certa directory siano password-protected, utilizzando HTTP Basic authentication

Programmazione Web

Programmare CGI in Python

Outline

- Il modulo Python “`cgi`”
  - Astrazione su GET vs POST
  - Query string decoding
  - Python-ic interface, basata su dizionari
  - Supporto per multiple values
- Gestione delle eccezioni, modulo “`cgit`”
- Gestione dei cookies, modulo “`Cookie`”
- Slide *Using Python for CGI programming*, by Guido Van Rossum
  - Inframezzate, as “usual” ...

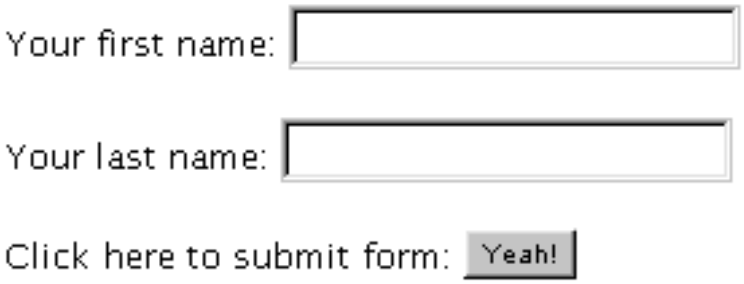
Using Python for CGI programming

Guido van Rossum
CNRI
(Corporation for National Research Initiatives, Reston, Virginia, USA)
guido@python.org
www.python.org

Outline

- HTML forms
- Basic CGI usage
- Setting up a debugging framework
- Security
- Handling persistent data
- Locking
- Sessions
- Cookies
- File upload
- Generating HTML
- Performance

A typical HTML form




```
<form method="POST" action="http://host.com/cgi-bin/test.py">
   <p>Your first name: <input type="text" name="firstname">
   <p>Your last name: <input type="text" name="lastname">
   <p>Click here to submit form: <input type="submit" value="Yeah!">
   <input type="hidden" name="session" value="1f9a2">
</form>
```

A typical CGI script

```
#!/usr/local/bin/python
import cgi

def main():
    print "Content-type: text/html\n"
    form = cgi.FieldStorage()          # parse query
    if form.has_key("firstname") and form["firstname"].value != "":
        print "<h1>Hello", form["firstname"].value, "</h1>"
    else:
        print "<h1>Error! Please enter first name.</h1>"

main()
```

CGI script structure

- Check form fields
  - use cgi.FieldStorage class to parse query
    - takes care of decoding, handles GET and POST
    - "foo=ab+cd%21ef&bar=spam" -->
      {'foo': 'ab cd!ef', 'bar': 'spam'} # (well, actually, ...)
- Perform action
  - this is up to you!
  - database interfaces available
- Generate HTTP + HTML output
  - print statements are simplest
  - template solutions available

Structure refinement

```
form = cgi.FieldStorage()
if not form:
    ...display blank form...
elif ...valid form...:
    ...perform action, display results (or next form)...
else:
    ...display error message (maybe repeating form)...
```

FieldStorage details

- Behaves like a dictionary:
  - .keys(), .has_key() # but not others!
  - dictionary-like object ("mapping")
- Items
  - values are MiniFieldStorage instances
    - .value gives field value!
  - if multiple values: *list* of MiniFieldStorage instances
    - if type(...) == types.ListType: ...
  - may also be FieldStorage instances
    - used for file upload (test .file attribute)

Other CGI niceties

- cgi.escape(s)
  - translate "<", "&", ">" to "<", "&", "&gt"
- cgi.parse_qs(string, keep_blank_values=0)
  - parse query string to dictionary {"foo": ["bar"], ...}
- cgi.parse([file], ...)
  - ditto, takes query string from default locations
- urllib.quote(s), urllib.unquote(s)
  - convert between "~" and "%7e" (etc.)
- urllib.urlencode(dict)
  - convert dictionary {"foo": "bar", ...} to query string "foo=bar&..." # note asymmetry with parse_qs() above

Dealing with bugs

- Things go wrong, you get a traceback...
- By default, tracebacks usually go to the server's error_log file...
- Printing a traceback to stdout is tricky
  - could happen before "Content-type" is printed
  - could happen in the middle of HTML markup
  - could contain markup itself
- What's needed is a...

Debugging framework

```
import cgi

def main():
    print "Content-type: text/html\n" # Do this first
    try:
        import worker      # module that does the real work
    except:
        print "<!-- --><hr><h1>Oops.  An error occurred.</h1>"
        cgi.print_exception() # Prints traceback, safely

main()
```

Gestione delle eccezioni: `cgit`

- I linguaggi di scripting solitamente sollevano eccezioni (+ traceback) quando qualcosa va storto
- Se ciò accade in un CGI program, è possibile che gli header non siano ancora stati generati → internal server error (causa “premature end of headers”)
- Il modulo Python `cgit` offre una soluzione

```
import cgitb; cgitb.enable()                # log su web
import cgitb; cgitb.enable(display=0,       # log su FS
                           logdir="/my/dir")
```

  - È una soluzione temporanea! Una volta reso più robusto il vostro CGI, si presuppone che l'uso di `cgit` venga commentato

Security notes

- Watch out when passing fields to the shell
  - e.g. os.popen("finger %s" % form["user"].value)
  - what if the value is "; cat /etc/passwd" ...
- Solutions:
  - Quote:
    - user = pipes.quote(form["user"].value)
  - Refuse:
    - if not re.match(r"^\w+$", user): ...error...
  - Sanitize:
    - user = re.sub(r"\W", "", form["user"].value)

Using persistent data

- Store/update data:
  - In plain files (simplest)
    - FAQ wizard uses this
  - In a (g)dbm file (better performance)
    - string keys, string values
  - In a "shelf" (stores objects)
    - avoids parsing/unparsing the values
  - In a real database (if you must)
    - 3rd party database extensions available
    - not my field of expertise

Plain files

```
key = ...username, or session key, or whatever...
try:
    f = open(key, "r")
    data = f.read()                 # read previous data
    f.close()
except IOError:
    data = ""                       # no file yet: provide initial data
data = update(data, form)           # do whatever must be done
f = open(key, "w")
f.write(data)                       # write new data
f.close()
# (could delete the file instead if updated data is empty)
```

(G)DBM files

```
# better performance if there are many records

import gdbm
key = ...username, or session key, or whatever...
db = gdbm.open("DATABASE", "w")          # open for reading+writing
if db.has_key(key):
    data = db[key]                       # read previous data
else:
    data = ""                            # provide initial data
data = update(data, form)
db[key] = data                           # write new data
db.close()
```

Shelves

```
# a shelf is a (g)dbm files that stores pickled Python objects

import shelve
class UserData: ...
key = ...username, or session key, or whatever...
db = shelve.open("DATABASE", "w")          # open for reading+writing
if db.has_key(key):
    data = db[key]            # an object!
else:
    data = UserData(key)    # create a new instance
data.update(form)
db[key] = data
db.close()
```

Locking

- (G)DBM files and shelves are not protected against concurrent updates!
- Multiple readers, single writer usually OK
  - simplest approach: only lock when writing
- Good filesystem-based locking is *hard*
  - no cross-platform solutions
  - unpleasant facts of life:
    - processes sometimes die without unlocking
    - processes sometimes take longer than expected
    - NFS semantics

A simple lock solution

```
import os, time

class Lock:

    def __init__(self, filename):
        self.filename = filename
        self.locked = 0

    def lock(self):
        assert not self.locked
        while 1:
        try:
            os.mkdir(self.filename)
            self.locked = 1
            return          # or break
        except os.error, err:
            time.sleep(1)
```

```
    def unlock(self):
        assert self.locked
        self.locked = 0
        os.rmdir(self.filename)

    # auto-unlock when lock object is deleted
    def __del__(self):
        if self.locked:
            self.unlock()


# for a big production with timeouts,
# see the Mailman source code (LockFile.py);
# it works on all Unixes and supports NFS;
# but not on Windows,
# and the code is very complex...
```

Gestione delle sessioni

- HTTP è stateless, manca il concetto di “sessione”
  - Non esiste modo nel protocollo di correlare: 2 richieste tra loro, ne tantomeno di associare “utenti” univoci a richieste
- Vari approcci per implementare le sessioni su HTTP
  - Supporto specifico in linguaggi di web-scripting (e.g., PHP)
  - Server-side: alla prima richiesta il CGI genere un id. univoco di sessione. A tutte le richieste successive l'id viene passato come parametro hidden

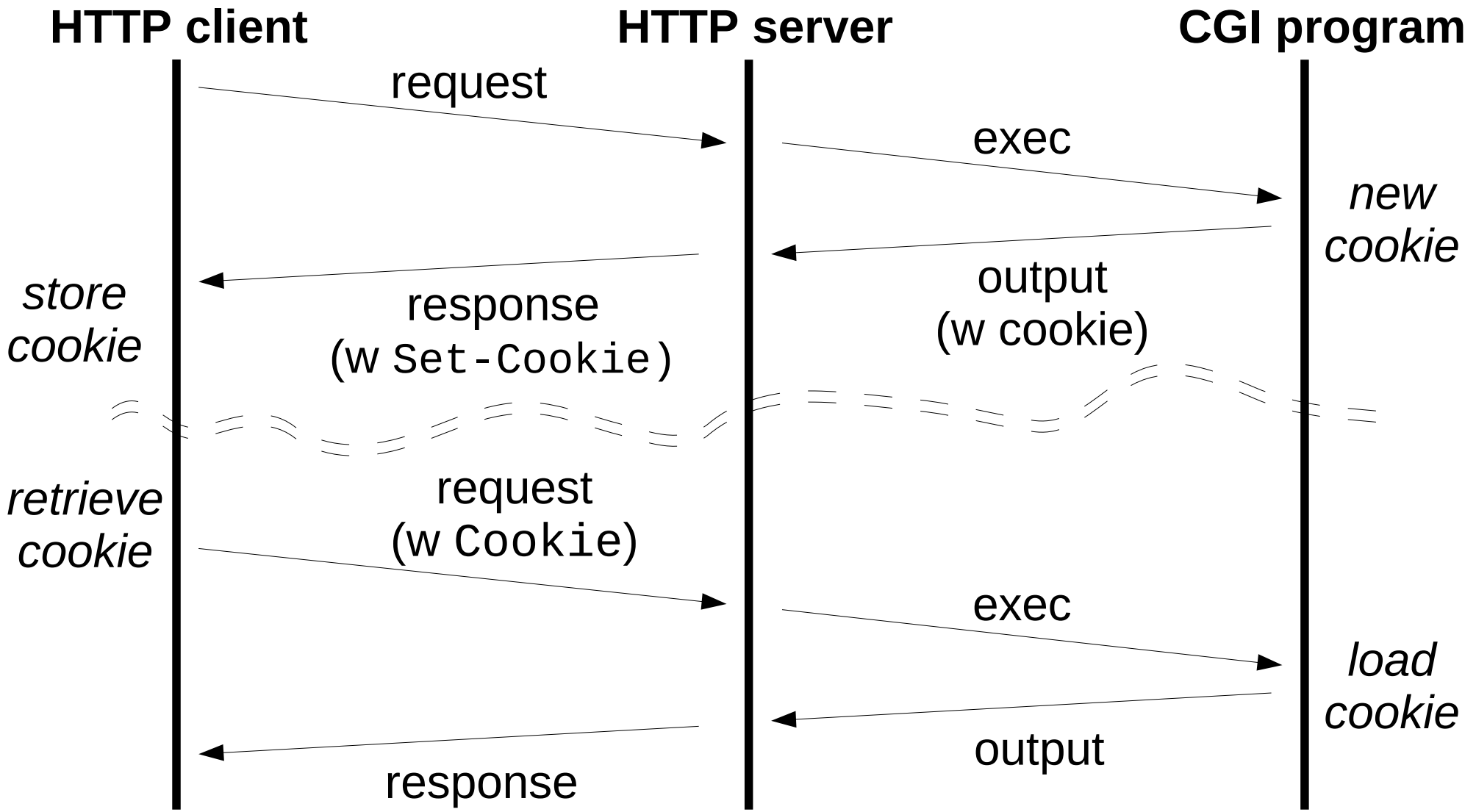
```
<input type=”hidden” name=”sid” value=”187afde5ed” />
```

  - Client-side: cookies

Cookies

- Un cookie è una “piccola” quantità di informazioni mantenuta nei client HTTP (e.g., browser) ed inviata a server HTTP assieme alla richiesta
- Il server richiede al client di memorizzare un cookie “a futura memoria” per creare una nuova sessione. Il cookie contiene un identificativo di sessione
  - Header su risposta HTTP Set-Cookie
- Ad ogni connessione successiva, il client re-invia il cookie, permettendo alle applicazioni CGI di correlare richieste pertinenti alla stessa sessione
  - Header su richiesta HTTP Cookie
- Nota: il concetto di sessione è vago, in pratica qualsiasi tipo di informazione può essere memorizzata in un cookie
  - Modulo limitazioni di spazio dovuti alla lunghezza tipica degli header
- Reference: RFC2109

Cookie scenario



Cookie's meta-information

- `Cookie` viene inviato o meno sulla basa di varie info
  - Server, età del cookie, nome del documento, …
  - Non possiamo inviare tutti i cookie a tutti i server! (privacy)
- Meta informazioni associate ad ogni cokie
  - *Domain*: nome di dominio di validità del cookie
  - *Comment*: descrizione
  - *Path*: URL di validità del cookie
  - *Max-Age*: expire time del cookie (in secondi)
  - *Secure*: richiede HTTPS
  - *Version*: versione della specifica cui il cookie è conforme

Cookie e CGI

- Come leggo l'header `Cookie` da un CGI program?
  - Variabile d'ambiente `HTTP_COOKIE`
- Come invio un cookie al browser client da un CGI program?
  - Inviando l'header `Set-Cookie` prima di inviare il BODY

Python's `Cookie`

- Il modulo Python `Cookie`
  - Implementa i cookie astrandoli come dizionari
  - Tutte le info nel dizionario sono inviate/ricevute via cookie
  - I valori del dizionario sono ogetti “Morsel” alle quali possono essere associate le meta-informazioni (e.g., expire)
- Nota di sicurezza
  - La classe legacy “Cookie” usa pickling per memorizzare le informazioni
  - Fare un-pickling di dati provenienti dalla rete è unsafe
  - SimpleCookie è da preferire, usa normali stringhe come rappresentazioni di valori
- Per un esempio si veda “Using Python for CGI programming”, slide 21

Cookie example

```
import os, cgi, Cookie

c = Cookie.Cookie()
try:
    c.load(os.environ["HTTP_COOKIE"])
except KeyError:
    pass

form = cgi.FieldStorage()
try:
    user = form["user"].value
except KeyError:
    try:
        user = c["user"].value
    except KeyError:
        user = "nobody"

c["user"] = user

print c

print """
<form action="/cgi-bin/test.py"
      method="get">
<input type="text" name="user"
      value="%s">
</form>
""" % cgi.escape(user)

# debug: show the cookie header we wrote
print "<pre>"
print cgi.escape(str(c))
print "</pre>"
```

File upload example

```
import cgi
form = cgi.FieldStorage()
if not form:
    print """
    <form action="/cgi-bin/test.py" method="POST" enctype="multipart/form-data">
    <input type="file" name="filename">
    <input type="submit">
    </form>
    """
elif form.has_key("filename"):
    item = form["filename"]
    if item.file:
        data = item.file.read()          # read contents of file
        print cgi.escape(data)           # rather dumb action
```

Generating HTML

- HTMLgen (Robin Friedrich)

  http://starship.python.net/crew/friedrich/HTMLgen/html/main.html

  ```
  >>> print H(1, "Chapter One")
  <H1>Chapter One</H1>
  >>> print A("http://www.python.org/", "Home page")
  <A HREF="http://www.python.org/">Home page</A>
  >>> # etc. (tables, forms, the works)
  ```

- HTMLcreate (Laurence Tratt)

  http://www.spods.dcs.kcl.ac.uk/~laurie/comp/python/htmlcreate/

  - not accessible at this time

CGI performance

- **What causes slow response?**
  - One process per CGI invocation
    - process creation (fork+exec)
    - Python interpreter startup time
    - importing library modules (somewhat fixable)
  - Connecting to a database!
    - this can be the killer if you use a real database
  - Your code?
    - probably not the bottleneck!

Avoiding fork()

- Python in Apache (mod_pyapache)
  - problems: stability; internal design
  - advantage: CGI compatible
  - may work if CGI scripts are simple and trusted
  - doesn't avoid database connection delay
- Use Python as webserver
  - slow for static content (use different port)
  - advantage: total control; session state is easy
- FastCGI, HTTPDAPI etc.
- ZOPE

ZOPE

- Z Object Publishing Environment
  - http://www.zope.org
  - complete dynamic website management tool
    - written in cross-platform Python; Open Source
  - http://host/path/to/object?size=5&type=spam
    - calls path.to.object(size=5, type="spam")
  - DTML: templatized HTML (embedded Python code)
  - ZOBD (Z Object DataBase; stores Python objects)
    - transactionsm selective undo, etc.
  - etc., etc.

Esercizi

- Effettuare il “porting” del wiki precedentemente realizzato al modulo Python `cgi`
- Estendere il wiki con le seguenti funzionalità
  - Aggiungere il supporto per file attachment
  - Implementare autenticazione basata su cookies
    - Nessun utente non autenticato può modificare le pagine
    - Dopo il login: le credenziali di accesso sono salvata in un cookie che viene usato per accessi successivi
    - Per semplicità: ignorate la registrazione, il wiki legge lo user database da un file su disco
  - Ci sono problemi di locking (accessi concorrenti che possono “pestarsi i piedi”) nel wiki?
    - Se si, come pensate di risolverli? .... ora risolveteli! :-)