Una implementazione di WSGI per Nginx

Manlio Perillo

10 Maggio 2008

Organizzazione del talk

Il talk sarà diviso in 4 parti:

- La prima tratterà di `WSGI`.
- La seconda parte tratterà di `Nginx`.
- La terza parte tratterà dell'implementazione di `WSGI` per Nginx (`mod_wsgi`)
- La quarta parte, infine, tratterà di `wsgix`, framework disegnato per poter sviluppare applicazioni che possano trarre vantaggio dalle caratteristiche di `mod_wsgi` per Nginx.

WSGI

- WSGI è definito nel PEP 333
  http://python.org/dev/peps/pep-0333/
- WSGI è un acronimo per: Python Web Server Gateway Interface.
- La versione attuale è la 1.0, ma è in discussione una nuova revisione, con importanti cambiamenti (ed il supporto a Python 3.x).
- WSGI ha ormai 4 anni (il PEP 333 è stato rilasciato il 07 Dicembre 2003)

WSGI: cosa è

- Come indicato nel nome, WSGI è una specifica che definisce come deve comportarsi un software che deve fare da gateway tra un server web ed una applicazione (o framework) Python.
- Il suo scopo primario è fornire una interfaccia standard che permetta ad una applicazione di non dipendere dal particolare server utilizzato (Nginx, Apache, Twisted, Base HTTP Server).
- WSGI permette anche lo sviluppo di applicazioni particolari, middeware, che si pongono tra il gateway WSGI e l'applicazione.
- I middleware sono stati spesso scritti male: http://dirtsimple.org/2007/02/wsgi-middleware-considered-harmful.html

WSGI: cosa non è

- WSGI non è un framework per lo sviluppo di applicazioni web.
- La sua interfaccia è a basso livello, basata su CGI.

Veloce panoramica su WSGI

- Lo scheletro di una applicazione *WSGI*:

```
def application(environ, start_response):
    """Simple application"""
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain')]
    start_response(status, response_headers)
    return ['Hello world!\n']
```

Veloce panoramica su WSGI

- Un esempio più complesso:

```
def application(environ, start_response):
    """Advanced application."""
    response_headers = [('Content-Type', 'text/plain')]

    input = environ['wsgi.input']
    log = environ['wsgi.errors']

    start_response('200 OK', response_headers)

    for line in input:
        log.write(repr(line))
        yield line
```

Veloce panoramica su WSGI

- Le maggiori differenze con CGI sono:
  - Il dizionario di ambiente (`environ`) è locale alla funzione, mentre con CGI le "`meta-variables`" sono globali (in `os.environ`)
  - Per inviare gli headers si deve invocare la funzione `start_response`.
  - L'applicazione deve restituire un oggetto iterabile, quindi può restituire un generatore.
- La possibilità di restituire un generatore, e di poter chiamare `start_response` all'interno del generatore permette di supportare server asincroni, ed anche applicazioni asincrone (benchè non previsto dagli autori di WSGI 1.0!).

Veloce panoramica su WSGI

- Nella prossima versione di WSGI (2.0) non ci sarà più `start_response`, questo per facilitare la scrittura di middlewares (**MA** perdiamo il supporto alle applicazioni asincrone).
- Ecco una semplice applicazione che usa WSGI 2.0:

```
def application(environ, start_response):
    """Advanced application with WSGI 2.0"""
    def app_iter():
        for line in input:
            log.write(repr(line))
            yield line

    response_headers = [('Content-Type', 'text/plain')]

    input = environ['wsgi.input']
    log = environ['wsgi.errors']

    return '200 OK', response_headers, app_iter()
```

Veloce introduzione a Nginx

- Nginx è un server HTTP (ma non solo!)
- Nginx usa il modello a eventi (asincrono).
  Sono supportati `select`, `poll`, `epoll`, `kqueue`, `rtsig`, `\dev\poll`, `eventport`.
- Nginx supporta SMP (Symmetric multiprocessing), il modello usato è quello di un processo master (supervisore) e un numero fisso di processi worker, ognuno dei quali chiama `accept` sullo stesso socket.
- E' importare non usare un numero troppo alto di processi worker. Tuttavia grazie al supporto a SMP, è possibile eseguire (con qualche limitazione) all'interno di Nginx applicazioni *sincrone*.

Veloce introduzione a Nginx

- Nginx può essere considerato un framework per lo sviluppo di server asincroni.
- Il server HTTP è solo un *modulo*.
  La distribuzione ufficiale di Nginx offre anche un proxy POP3 e IMAP4.
- Nginx offre un API di medio livello sopra il sistema operativo, che permette di scrivere moduli senza preoccuparsi di troppi dettagli.
- Ma a differenza di un framework come `Twisted`, alcune parti diverse di codice sono fortemente legate tra di loro.

Veloce introduzione a Nginx

- Nginx offre anche supporto per l'allocazione della memoria tramite *pools*.
  Questo permette di non doversi (quasi) mai preoccuparsi di liberare la memoria.
  Nel modulo HTTP c'è un *pool* allocato per ogni request, e tale pool è deallocato quando la request termina.
- Nginx offre una sintassi per la configurazione molto flessibile e facile da usare.
- Il codice di Nginx è scritto molto bene e contiene molte piccole cose che sono di aiuto per chi scrive moduli.
- Nginx ha anche un mini linguaggio di scripting (abbastanza limitato).
  Forse verrà rimpiazzato da Lua.

Storia di Nginx

- In spring 2001 Igor wrote Apache mod_accel that is enhanced replacement of mod_proxy.
- Spring 2002 - first drafts.
- 04.10.2004 - 0.0.1 public release (BTW, it was 47 Sputnik anniversary).
- `http://article.gmane.org/gmane.comp.web.nginx.english/4824`

Storia di Nginx

- Ad oggi, Nginx è sviluppato da una sola persona (Igor Sysoev).
- Igor, purtroppo, non conosce benissimo l'inglese.
- Il repository di Nginx (Subversion) non è pubblico.
- Nel Settembre 2006 è stato reso disponibile il wiki non ufficiale in inglese, con la traduzione della documentazione di Nginx dal russo, ad opera di Aleksandar Lazic e Cliff Wells.
- Nginx è ancora in fase di sviluppo, ad oggi la versione stabile è la 0.6.30.
- La mancanza di una versione **1.0** non ha ostacolato la rapida diffusione di Nginx!

Introduzione

- Perchè sviluppare un nuova nuova implementazione di WSGI?
- Le varie soluzioni sviluppate in Python *puro* sono facili da installare ma non sono molto adatte per un ambiente di produzione.
  Oltre ai noti problemi con il supporto robusto alla concorrenza, molte di queste soluzioni non permettono, ad esempio, il virtual hosting.
- Twisted è una valida soluzione, ma Twisted Web 2 non è ancora pronto.
- Tutte le soluzioni scritte in Python usano un singolo processo e questo complica la gestione del reloading dell'applicazione.

Introduzione

- Apache con il suo `mod_wsgi` è una soluzione adatta per un ambiente di produzione, ma potrebbe richiedere risorse non indifferenti.
- Le compagnie che offrono soluzioni per hosting condiviso e non vogliono usare Apache sono "costrette" ad usare `flup` che non ha prestazioni ottimali.
- In realtà il modulo WSGI per Apache è molto flessibile, in quanto permette di eseguire le applicazioni Python in un pool di processi separato dal core di Apache.

Introduzione

- Il modulo WSGI per Nginx permette un deployment rapido e si basa su un server ben noto per la sua stabilità ed efficienza.
- Possibilità di eseguire più di una applicazioni all'interno di Nginx.
- Possibilità di avere un controllo completo e immediato su vari aspetti come il logging, tramite una sintassi per la configurazione molto flessibile.
- Ma uno dei motivi principali per lo sviluppo è stato sperimentare con la definizione ed implementazione di estensioni a WSGI per il supporto di applicazioni asincrone.

Il codice

- Il codice del modulo è disponibile, al momento, solo come repository Mercurial: http://hg.mperillo.ath.cx/nginx/mod_wsgi/
- Con il rilascio della prossima versione il codice verrà spostato su un nuovo sito, insieme ad un bug tracker e mailing list.
  Non è stato ancora deciso nulla a riguardo.
- Nel file `README` sono presenti le istruzioni per compilare il modulo e la documentazione delle direttive supportate.

Configurazione e setup

- In Nginx i moduli vanno compilati staticamente!
- mod_wsgi è stato testato con Nginx 0.5.35.
- Da pochi giorni, la nuova versione stabile di Nginx è la 0.6.30.
- A causa di alcune modifiche al codice, mod_wsgi necessita di una patch per poter essere compilato con il branch 0.6.x.
- Nel codice di Nginx non è definita una macro contenente la versione, quindi non è possibile effettuare una compilazione condizionata!

Configurazione e setup

- Dalla directory che contiene i sorgenti di Nginx:

```
$ ./configure --add-module=/path/to/mod_wsgi/ \
    --with-debug
```

- `$ make`
- Come root:

```
$ make install
```

  Di default, i files vengono copiati in `/usr/local/nginx`

- Nella distribuzione di `mod_wsgi` ci sono altri files che devono essere installati:

```
$ python setup.py install
```

Alcune informazioni sullo sviluppo

- Il modulo WSGI per Nginx non avrebbe mai visto la luce se non ci fosse stato mod_wsgi per Apache (di Graham Dumpleton), da usare come riferimento!
- La prima versione forniva una implementazione della bozza di WSGI 2.0, dato che il codice era più facile e lineare da scrivere senza `start_response`.
- L'idea originale era di usare un solo interprete per server.
- Purtroppo però molte applicazioni (**Django**) fanno uso di uno stato globale, e quindi ci può essere solo una applicazione per interprete.
- E' stato quindi aggiunto il supporto a interpreti multipli (ma non è il comportamento di default).

Alcune informazioni sullo sviluppo

- Python offre supporto per multipli interpreti, ma non è implementato bene e diversi moduli hanno problemi quando girano con un ambiente multi-interprete.
- Per maggiori informazioni si può consultare l'ottima guida sul wiki di mod_wsgi per Apache:
  `http://code.google.com/p/modwsgi/wiki/ApplicationIssues#Mult`
- Un altro problema quando si usano i sotto interpreti è che la loro finalizzazione non è *pulita*.
  - Non viene eseguita la funzione registrata con `sys.atexitfunc`.
  - Non vengono invocati i distruttori delle variabili globali.

Alcune informazioni sullo sviluppo

- Per fortuna alcune di queste limitazioni possono essere risolte via codice.
- Il modulo WSGI esegue correttamente `sys.exitfunc` quando ciascun sotto interprete viene finalizzato.
- Il modulo WSGI **non** esegue i distruttori delle variabili globali, a causa di un bug.
- Grazie al cielo, Nginx non usa il multithread, altrimenti la stabilità mentale dell'autore sarebbe stata a rischio.

Avvertimendi d'uso

- Il modulo WSGI per Nginx è molto efficiente, sia in termini di consumo CPU che di consumo di memoria, oltre ad avere un relativamente basso impatto sul carico del sistema, anche con alti livelli di concorrenza.
- Ovviamente se volete eseguire una *grossa* applicazione scritta in Django, ad esempio, le prestazioni del modulo WSGI avranno poca influenza rispetto alle prestazioni della vostra applicazione, specialmente se largamente I/O bound.
- Se ne avete la possibilità fate sempre dei test comparativi con Apache.

Deployment

- Il modulo WSGI per Nginx **non** è un rimpiazzo per `mod_wsgi` di Apache.
- Il modulo WSGI per Nginx **è** un rimpiazzo per `flup` e ogni altro server HTTP/SCGI/FastCGI scritto in Python (con l'eccezione, ovviamente, di `Twisted`).
- Il modulo WSGI per Nginx **non** è un modulo di "prima classe" in Nginx, dato che "prende il controllo" dell'intero server.
  Il supporto a WSGI è stato sviluppato come modulo per evitare di dover implementare un server HTTP da zero.

Deployment

Un esempio di file di configurazione:

```
http {
    include       conf/mime.types;
    default_type  application/octet-stream;

    server {
        listen       8080;
        server_name  localhost;

        wsgi_middleware  wsgiref.validate  validator;

        location / {
            wsgi_pass /usr/local/nginx/test1.py;

            include conf/wsgi_vars;
            # override existing HTTP_ variables
            wsgi_var HTTP_COOKIE  $http_cookie;

            wsgi_script_reloading on;
        }
    }
}
```

- Una prima funzionalità distintiva è che è possibile definire middleware direttamente dalla configurazione di Nginx.
- Questo rende la configurazione più facile da gestire, specialmente da chi non è esperto di Python e deve semplicemente installare una applicazione.
- Il modulo WSGI mira ad essere una soluzione completa per il deployment, in modo analogo a `Paste` (ma usando un approccio molto diverso).
- La possibilità di definire i middleware in un file di configurazione e non dal codice Python pone problemi interessanti.

Deployment

- Questo è un esempio di come è definito un middleware in Paste:

```
def auth_filter_factory(global_conf, req_usernames):
    # space-separated list of usernames:
    req_usernames = req_usernames.split()

    def make_filter(app):
        return AuthFilter(app, req_usernames)

    return make_filter

class AuthFilter(object):
    def __init__(self, app, req_usernames):
        self.app = app
        self.req_usernames = req_usernames

    def __call__(self, environ, start_response):
        if environ.get('REMOTE_USER') in self.req_usernames:
            return self.app(environ, start_response)
        start_response(
            '403 Forbidden', [('Content-type', 'text/html')])
        return ['You are forbidden to view this resource']
```

Deployment

- L'infrastruttura dei middleware in Paste non è riutilizzabile se vogliamo definire i middleware in Nginx.
- Un middleware non può/deve(?) essere inizializzato tramite una funzione factory.
- Invece un middleware dovrebbe leggere le variabili di configurazione dal dizionario di `environ`.
- In particolare, tutto lo stato del middleware (ma anche delle applicazioni) dovrebbe essere memorizzato nel `environ`.
- Se è necessario mantenere un stato (ad esempio per il caching), allora deve essere fatto in modo trasparente (il comportamente del middleware deve essere lo stesso come se fosse stateless).

Deployment

- Esempio di middleware sviluppato nel modo *giusto*:

```
_req_usernames_cache = {}
class auth_filter(object):
    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        # space-separated list of usernames:
        key = environ['app.req_usernames']

        req_usernames = _req_usernames_cache.get(key)
        if req_usernames is None:
           req_usernames = key.split()
           cache[key] = req_usernames

        if environ.get('REMOTE_USER') in req_usernames:
            return self.app(environ, start_response)
        start_response(
            '403 Forbidden', [('Content-type', 'text/html')])
        return ['You are forbidden to view this resource']
```

Deployment

- Un altra caratteristica particolare è come sono gestite le variabili di `environ`.
- Il modulo WSGI non aggiunge le variabili via codice, ma lascia all'utente la possibilità di definire **tutte** le variabili da Nginx.
- Ovviamente l'eccezione è con le variabili specifiche di `WSGI`, come wsgi.input e wsgi.errors.
- La versione corrente del modulo aggiunge anche le variabili SCRIPT_NAME a PATH_INFO, perchè Nginx non definisce queste variabili.
- In una delle prossime versioni del modulo WSGI queste due variabili saranno disponibili come variabili Nginx.
  `$wsgi_script_name`, `$wsgi_path_info`

Deployment

- Tutte le variabili richieste da WSGI (e CGI) sono definite nel file `wsgi_vars`.
  Tale file è installato dallo script `setup.py`.
- Nell'`environ` vengono prima inserite le variabili specifiche WSGI, quindi gli headers della request HTTP ed infine le variabili definibili dall'utente.
- Questo significa che possiamo modificare il valore di alcuni headers.
- Nell'esempio facciamo questo per ridefinire `HTTP_COOKIE`.
- Nginx **non** combina gli headers multipli (`http://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html#sec4.`
  I cookie sono comunque un caso anomalo...

Deployment

- Nginx supporta diversi segnali per il controllo del server.
- Una feature interessante è la possibilità di aggiornare Nginx ad una nuova versione senza doverlo arrestare.
- Dato che le request HTTP sono gestire dai processi worker, è possibile arrestare questi processi (sia in modalità aggraziata che veloce).

Deployment

- Quando Nginx riceve il segnale `HUP` rilegge il file di configurazione e, se è corretto, riavvia i processi worker, terminandoli in modalità agraziata.
- Questo significa che se avete, ad esempio, 4 worker, per un certo intervallo di tempo avrete in totale 9 processi attivi!
- Il modulo WSGI per Nginx permette di ricaricare l'applicazione WSGI quando il modulo principale viene modificato e se l'opzione `wsgi_script_reloading` è abilitata.
- E' possibile sia effettuare un semplice reloading del modulo (soluzione adottata da altre implementazioni in Python).
- Oppure è possibile riavviare il processo worker, in modo che non ci sono problemi con le dipendenze.

Deployment

- La request corrente viene portata a termine usando il "vecchio" "stato", ma viene aggiunto l'header "`Refresh: 0`" in modo che il client possa rieffettuare la request per ottenere la versione della risorsa aggiornata.
- Questa soluzione non è ottimale.
- L'header `Refresh` è non standard ed obsoleto.
- E' preferibile inviare un segnale `HUP` "a mano".

Supporto a funzionalità avanzate

- Il modulo WSGI per Nginx implementa `wsgi.file_wrapper`
- L'implementazione è molto efficiente, specialmente se `sendfile` è abilitato in Nginx.
- E' allo studio la possibilità di implementare il supporto per le *subrequest* di Nginx.
- Questo permetterebbe di assemblare una pagina da diverse sorgenti (come SSI) in modo **molto** efficiente.

Supporto a funzionalità avanzate

- L'implementazione di WSGI per Nginx offre del supporto integrato per alcune funzionalità "avanzate" di HTTP, in modo del tutto trasparente all'applicazione.
- Se l'opzione `wsgi_allow_ranges` è attiva, Nginx penserà a gestire le request che richiedono parti parziali della risposta.
- La versione attuale di Nginx non supporta questa funzionalità se l'applicazione genera la risposta in modo generativo. XXX
- Inoltre l'applicazione deve comunque generare tutta la risposta.
- Questo non è un problema, l'importante è risparmiare la banda.

Supporto a funzionalità avanzate

- Il modulo WSGI per Nginx offre anche il supporto per la validazione della copia in cache del client, tramite l'header `Last-Modified`.
- Al momento non è possibile disabilitare questa funzionalità.
- Se l'applicazione usa un generatore, il modulo WSGI è in grado di ottimizzarne l'esecuzione se la request è `HEAD` o se il client ha una copia aggiornata in cache.
- In pratica dopo aver inviato gli headers, il resto della applicazione non ha bisogno di essere eseguita.
- Si veda la documentazione dell'opzione `wsgi_optimize_send_headers` per maggiori informazioni.

WSGI e server asincroni

- Benchè il PEP che definisce WSGI insiste spesso sull'uso dei thread, il protocollo è stato pensato anche per supportare server asincroni.
- Consideriamo la seguente applicazione:

```
def application(environ, start_response):
    """Advanced application: file streaming."""
    response_headers = [('Content-Type', 'text/plain')]

    f = file('/some/big/text/file', 'r')
    start_response('200 OK', response_headers)

    for line in file:
        yield line
```

- Un server asincrono non può inviare un flusso continuo di dati verso il client, dato che la scrittura sul socket è non bloccante.

WSGI e server asincroni

- La *chiave* è il supporto ai generatori.
- Infatti un gateway può sospendere l'esecuzione dell'applicazione quando l'applicazione fa uno `yield`.
- `mod_wsgi`, quindi, sospende l'esecuzione dell'applicazione nel caso in cui il la stringa restituita da `yield` non può essere inviata completamente al buffer del sistema operativo.

WSGI e server asincroni

- Purtroppo per motivi di compatibilità WSGI supporta un approccio differente per inviare dati al client.
- La funzione `start_response` restituisce una funzione `write` che, quando invocata, invia la stringa passata al client.
- La funzione `write` **non** può essere implementata come richiesto da WSGI in un server asincrono (in particolare per un server asincrono che non usa threads).

WSGI e server asincroni

- Il modulo WSGI per Nginx implementa due versioni della funzione `write`.
- La prima versione usa un hack: mette il socket usato da Nginx temporaneamente in modalità sincrona, in modo che `write` non ritorna finchè tutti i dati sono stati inviati nel buffer del sistema operativo.
- Questa soluzione rispetta la specifica per WSGI, ma non è l'ideale per Nginx!
- La seconda versione bufferizza tutti i dati in un buffer temporaneo, e li invia al client quando il controllo ritorna a Nginx.
- Questa soluzione non rispetta la specifica per WSGI.

WSGI ed applicazioni asincrone

- Quando si parla di supporto ad applicazioni asincrone la situazione è più complessa.
- WSGI, in teoria, supporta applicazioni asincrone, ma questa possibilità è stata a lungo trascurata.
- Il motivo è che per supportare applicazioni asincrone abbiamo bisogno di introdurre il concetto di continuazione (Continuation passing style).
- Ad esempio Twisted ha le Deferred.
- Non esiste un accordo generale su come debba essere una applicazione asincrona.

WSGI ed applicazioni asincrone

- Il modulo WSGI per Nginx implementa delle estensioni **sperimentali** per offire supporto base alle applicazioni asincrone.
- L'interfaccia usata si basa su quella di `poll` e permette di registrare un socket nel reattore di Nginx.
- Per prima cosa, dato un socket che vogliamo registrare, dobbiamo creare un wrapper con:

```
s = socket(...)

c = environ['ngx.connection_wrapper'](s)
```

WSGI ed applicazioni asincrone

- Una volta ottenuto un wrapper per la connessione, possiamo registrarlo.

```
environ['ngx.poll_register](c, WSGI_POLLIN)
```

- Quindi per sospendere l'esecuzione dell'applicazione fino a quando la connessione è pronta a leggere dei dati:

```
timeout = 3000
state = environ['ngx.poll](timeout)

yield ''
c, flags = state()
    if c is None:
        start_response('500 ERROR', headers)
        yield timedout'
        return
```

WSGI ed applicazioni asincrone

- E' importante avere presente che non è `environ['ngx.poll](timeout)` a sospendere l'esecuzione.
- L'esecuzione viene sospesa quando, tramite lo `yield` il controllo ritorna a Nginx.
- Questo permette di supportare applicazioni asincrone restando compatibilità con WSGI.
- Tuttavia l'interfaccia non è molto flessibile, specialmente se `ngx.poll` deve essere chiamata in una funzione separata e non all'interno della funzione eseguita da Nginx.
- Infatti è necessario che tutte le funzioni nella "catena di chiamate" alla nostra funzione effettuino uno `yield`.

WSGI ed applicazioni asincrone

- Infine un ulteriore problema è che una applicazione "normale" deve essere riscritta, anche se non in modo radicale.
- Al momento sono pochissime le librerie che forniscono un API che permette di integrarsi con un reattore esterno.
- PostgreSQL e curl, che per fortuna coprono l'80% dei bisogni.
- Esempio:
  `http://hg.mperillo.ath.cx/nginx/mod_wsgi/file/tip/examples/nginx-postgres-async.py`
  Applicazione WSGI asincrona che effettua una query ad un database PostgreSQL

WSGI e coroutines

- E' possibile migliorare il supporto per l'esecuzione di applicazioni senza bloccare Nginx sulle operazioni di I/O?
- Il concetto base è quello dei micro threads o meglio, coroutines.
- Il primo supporto alle coroutines in Python si è avuto con Stackless.
- Stackless non ha avuto una calda accoglienza da parte della comunità Python.
- Stackless è un fork di CPython.
- Fortunatamente il supporto alle coroutines è disponibile tramite `greelets`, parte di `py lib`
  `http://codespeak.net/py/dist/greenlet.html`

WSGI e coroutines

- greenlets permette di creare delle coroutines all'interno dell'interprete standard CPython.
- L'implementazione si basa su quella di Stackless, ma greenlets fornisce solo il supporto base.
- Nel nostro caso questa interfaccia di basso livello è tutto quello che ci occorre.
- Non abbiamo bisogno del supporto allo scheduling, dato che ci pensa Nginx.
- Non abbiamo bisogno di canali di comunicazione (e relativa sincronizzazione) dato che HTTP è REST full. (XXX check me)

WSGI e coroutines

- Con il supporto integrato a greenlets nel modulo WSGI di Nginx, l'applicazione WSGI sarà eseguita in un greelet.
- In questo modo una qualsiasi funzione è in grado di sospendere direttamente l'applicazione corrente.
- Lo switch tra applicazioni potrà essere gestito dal codice utente, e le funzioni come `ngx.poll` potranno essere fornite come modulo esterno, invece di essere "legate" alla request corrente. XXX
- E' possibile implementare una funzione `read(s, n)` che sospende l'applicazione del greenlet corrente fino a quando **esattamente** n bytes sono stati letti dal socket (o non ci sono più dati disponibili).

WSGI e coroutines

- L'integrazione con greelets sarà implementata dopo questa conferenza.
- Se il risultato è soddisfacente, le estensioni implementate attualmente per il supporto di applicazioni asincrone **potrebbero** essere rimosse.
- In teoria dovrebbe essere possibile far diventare non bloccanti applicazioni come Django!

Chi usa mod_wsgi per Nginx?

- Benchè il modulo sia ancora in fase di sviluppo (l'ultima versione è la 0.0.6) qualche intrepido utente lo sta cominciando ad usare.
- Nginx con mod_wsgi è disponibile su Unbit.
  `http://http://wiki.unbit.it/Nginx`.
- Nginx con mod_wsgi è supportato da Eventlet, di Donovan Preston.
  `http://wiki.secondlife.com/wiki/Eventlet/Documentation#Using_Eventlet_with_Nginx`.

Qualche extra

- Come detto, una delle richieste per il modulo WSGI per Nginx è la facilità di deployment.
- Nella directory `bin` ci sono alcuni script di supporto.
- Il primo script è `ngx_wsgi.py`, che è un wrapper, scritto in Python, per eseguire Nginx passandogli le opzioni dalla riga di comando.
- Il secondo script è `ngx_wsgiref.py` che, tramite `wsgiref` mette a disposizione un server di testing che presenta una interfaccia a riga di comando simile a quella di `ngx_wsgi.py`.
- Infine lo script `ngx_wsgitest.py` esegue una applicazione WSGI in un ambiente di test.

TODO

- Supporto per il logging (livelli di errore, etc)
- Supporto alla memoria condivisa.
- Cercare di spiegare chiaramente che il modulo WSGI per Nginx non è come `mod_wsgi` per Apache, è che una applicazione Python può bloccare un intero processo worker (quindi Nginx non è in grado si servire altre richieste, ad esempio a files statici).

Introduzione

Non solo ti sei messo ad implementare un gateway WSGI in C, hai pure implementato un framework?

- `wsgix` è stato progettato per poter scrivere applicazioni WSGI che possano girare in modo efficiente (e conveniente) in Nginx.
- La prima scelta di design fatta è che tutto lo stato, incluse le opzioni vanno nel dizionario `environ`.
- La seconda scelta è che l'API deve essere di basso livello. Dobbiamo **sempre** aver presente che stiamo scrivendo una applicazione web.
- Deve essere possibile poter usare, all'occorrenza, le estensioni asincrone per WSGI implementate in `mod_wsgi` di Nginx.

Introduzione

- `wsgix` è ancora in fase di sviluppo attivo, e diverse funzionalità non sono state ancora implementate o completate.
- `wsgix` è disponibile su repository Mercurial all'indirizzo: `http://hg.mperillo.ath.cx/wsgix/`
- `wsgix` è disponibile con licenza `MIT`.
- La documentazione è praticamente inesistente, ma con queste slides si spera di rimediare.
- Esiste una applicazione di esempio, disponibile su repository separato: `http://hg.mperillo.ath.cx/wsgix/examples/sample/`

Introduzione

- Abbiamo detto all'inizio del talk che WSGI non è pensato per scrivere applicazioni.
- Ma è davvero necessario introdurre astrazioni sopra WSGI, come gli oggetti Request e Response?
- Un esempio di libreria che implementa queste astrazioni è `webop`: `http://pythonpaste.org/webob/`
- La risposta è **NO**, non è necessario introdurre astrazioni sopra WSGI; `wsgix` ne è la prova.

Esempi

- `wsgix` contiene una applicazione completa per servire i file statici.
  Questa applicazione illustra tutti i concetti base di wsgix.
  `wsgix.contrib.static`
- `wsgix` implementa un middleware per gestire messaggi verso l'utente.
  Questo middleware illustra come sviluppare un middelware nel "modo corretto".
  `wsgix.contrib.messages`

Configurazione

- Come detto il dizionario `environ` è al centro di `wsgix`.
- Tutte le opzioni di configurazione per una applicazione sono memorizzate nell'`environ`.
- In questo modo non importa come queste opzioni vengono inserite nell'`environ`.
- Ad esempio è possibile definire tutte le opzioni di configurazione all'interno del file di configurazione di Nginx.
- Questa soluzione ha il vantaggio di essere facile da manutenere, specialmente da chi non conosce Python e deve semplicemente eseguire una applicazione.
- Lo svantaggio è che non si possono definire configurazioni complesse.

Configurazione

- Il modulo `wsgi.options` implememta il supporto per il parsing delle opzioni e per l'accesso a queste opzioni in una request.
- Il modulo wsgix.urldispatch implementa un semplice dispatching per le url, basato su regex.
- All'interno dell'`UrlDispatch` è possibile definire un ulteriore dizionario, i cui valori saranno inseriti nel dizionario environ.
- Il sotto package `wsgix.conf` implementa alcune funzioni di supporto per gestire la configurazione in formato YAML.
- La funzione

  `load_application_settings(application, name='settings.yml')` carica il file di configurazione relativo alla data applicazione, usando costruttori personalizzati per offrire maggiore flessibilità nella configurazione.

Configurazione

- Il sotto package `wsgix.conf` implementa anche un middleware per caricare la configurazione automaticamente.
- Non è obligatorio usare questo middleware e non è obligatorio utilizzare YAML.
- Potete usare il formato e le procedure che preferite, l'importante è che le opzioni siano memorizzate nell'`environ` secondo alcune convenzioni.
- La convenzione in uso in `WSGI` è di avere, ad esempio, opzioni come `wsgix.auth.session_lifetime`.
- Ossia usiamo la notazione "." per denotare una struttura gerarchica in una struttura flat.

URL dispatching

- L'URL dispatching è molto semplice.

```
dispatcher = urldispatch.UrlDispatch()
dispatcher.add('/login', application=views.do_login)
dispatcher.add('/', GET=views.do_login)
```

- L'url dispatching supporta l'estensione `wsgiorg.routing_args` `http://wsgi.org/wsgi/Specifications/routing_args`
- Dato che la definizione delle URL è il cuore di un applicazione, è importante che la configurazione per il dispatching abbia la migliore semantica possibile. XXX
- In futuro è possibile che a `wsgix` venga aggiunto un mini linguaggio per la gestione delle URL.

Autenticazione

- Implementare l'autenticazione correttamente è complesso.
- `http://www.berenddeboer.net/rest/authentication.html`
- Non reimplementare la ruota.
- Se possibile, evitare di usare i cookie (anche se a volte non ci sono altre soluzioni).
- Nel seguito considereremo solo l'autenticazione basata su password.
- Un veloce elenco di alcuni dei metodi di autenticazione:
  - HTTP Basic
  - HTTP Digest
  - Cookie di sessione
  - Cookie firmati

Autenticazione

- Il supporto per l'autenticazione è implementato usando i concetti di Twisted Cred.
- `wsgix` implementa due tipi di autenticazione.
- Il primo è basato sui cookie di sessione, e l'implementazione consiste in un middleware.
- Il middleware `wsxix.auth.middleware` è basato sulla implementazione del pattern Guard di Nevow.
- Il middelware supporta, in modo trasparente, l'autenticazione tramite HTML forms e la gestione della sessione autenticata tramite cookie.
- Ma supporta anche HTTP Basic.

Autenticazione con cookie di sessione

- A differenza di altre implementazioni, in `wsgix` consideriamo l'`ID` di sessione come credenziale.
- Altre implementazioni usano la sessione come un oggetto opaco dove sono memorizzare le vere credentiali da usare.
- L'implementazione corrente è strettamente legata al backend (database relazionale)
- L'autenticazione tramite cookie di sessione ha problemi di sicurezza (attacchi replica).
- Questi e altri problemi si possono risolvere in parte, ma non del tutto.
- Inoltre richiede la scrittura di codice custom, e l'autore non ne ha voglia!

Autenticazione con HTTP Digest

- HTTP Digest è una valida alternativa.
- Innanzitutto, grazie al supporto da parte del client, abbiamo una elevata protezione contro diversi attacchi contro i quali i cookie e HTTP Basic non offrono protezione.
- HTTP Digest è comunque debole contro attacchi MIT (Man in the Middle).
- Inoltre non protegge il contenuto delle pagine (ma non è per questo che è stato pensato).
- Per questi casi la soluzione è usare SSL.

Autenticazione con HTTP Digest

- HTTP Digest non è supportato completamente da tutti i client.
- Ci sono alcuni problemi con comportamente non previsti, ma fortunatamente facilmente risolvibili.
- Il problema più grande è la limitata usabilità.
- Le credenziali sono "chieste" direttamente dal browser con una finestra di dialogo molto spartana e su cui non abbiamo controllo.
- I browser (con l'eccezione di Firefox, ma solo con un plugin esterno) non offrono supporto per il logout.

Autenticazione con HTTP Digest

- `wsgix` implementa HTTP Digest tramite un middleware, però a differenza del supporto per l'autenticazione tramite cookie di sessione, il login ed il logout devono essere gestiti dal codice utente.
- `wsgix` offre supporto **sperimentale** (personalmente non ho visto altre implementazioni simili) per il logout "facile".
- In particolare, in fase di logout viene inviato al client un `nonce` speciale contenente all'inizio la stringa `logout`.
  Inoltre la risposta viene marcata come `stale` (si veda l'RFC 2617 per maggiori informazioni), in modo che il browser non chieda di nuovo la password all'utente, ma semplicemente rigeneri la sua risposta in basa al nuovo `nonce`.

Autorizzazione

- Al momento `wsgix` non offre supporto per l'autorizzazione.
- Il supporto è in fase di sviluppo.
- Come il resto del framework, le linee guida saranno quelle di essere a basso livello.
- Quindi niente sistema di permessi ad alto livello (vedi Django o Trac).
- Invece il sistema di autorizzazioni si baserà sulle url.
- Per ogni url e per ogni metodo HTTP (GET, POST, PUT, DELETE) andremo a definire gli utenti o i gruppi abilitati ad eseguire quella operazione.

Autorizzazione

- Probabilmente il supporto alle autorizzazioni sarà incluso nel mini linguaggio per la gestione dell'url dispatching.
- Se le autorizzazioni sono dinamiche, si può modificare il file di configurazione e inviare un segnale `HUP` ad Nginx.
- Situazioni più complesse dovranno essere sviluppate ad hoc (vedi ACL), ma mantenersi a basso livello significa avere una speranza di sviluppare una soluzione riutilizzabile.

Gestione stato lato client

- `wsgix` offre supporto per gestire lo stato lato client ad alto livello.
- Ovviamente lo stato è gestito tramite cookie.
- In futuro sarà aggiunto il supporto per firmare il cookie digitalmente, per assicurare l'integrità dei dati.
- L'API è molto semplice.
  Come la maggioranza delle altre funzioni in `wsgix`, le funzioni accettano l'environ e gli headers.
  Gli headers sono una instanza di `wsgiref.headers.Headers`.

Gestione stato lato client

- ```
  state_set(environ, headers, name, encoders=SCALAR_ENCODERS,
            **values)

  state_get(environ, name, scalar=False, decoders=SCALAR_DECODERS)

  state_del(environ, headers, name)
  ```
- Lo stato è gestito tramite un dizionario.
- Lo stato viene serializzato tramite urlencoding.
- Ogni valore scalare viene codificato secondo lo schema `t.value`.
- Ad esempio “`i.10`” rappresente l’intero 10.
  ```

Gestione forms

- `wsgix` offre supporto per la gestione dei forms HTML.
- Questo è l'unico caso in cui viene usata una astrazione ad "alto" livello.
- Ma questa astrazione è basata sulla specifica XForms. `http://www.w3.org/MarkUp/Forms/`
- Il supporto è quasi completo ed utilizzabile, ma al momento non sono stati scritti degli esempi.