

Un server in JavaScript grazie a node.js

Come usare questo linguaggio, ormai molto comune, per la programmazione applicativa.

Non capita spesso di sentire cercare sviluppatori JavaScript per applicazioni destinate a girare nel cloud o in una server room, ma potrebbe accadere perché una delle tecnologie più interessanti, e in crescita, è basata proprio su questo linguaggio. Si tratta di node.js, un'infrastruttura server che usa JavaScript per la programmazione applicativa. C'è una ragione strutturale per giustificare la scelta del linguaggio: node.js è progettata intorno a un'idea architetturale radicalmente diversa da qualunque altro ambiente server e JavaScript è un linguaggio di uso comune più adatti per calarsi sopra.

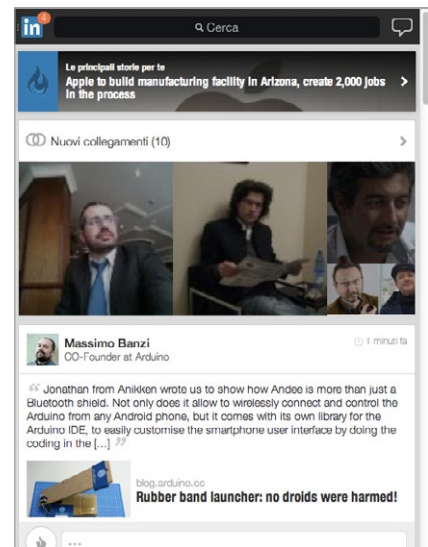
Un problema di efficienza

Qualsiasi server web, ha un modello di funzionamento semplice: attende richieste, le elabora, risponde qualcosa e ritorna alla casella numero uno. Tutto qui. Non potrebbe essere più lineare. Dove il discorso si complica è nell'efficienza del sistema, perché una soluzione brillante in laboratorio non necessariamente funziona per la home page della CNN, o di qualunque sito bersagliato da ben più di un accesso contemporaneo. Siti come Google o Facebook, infatti, devono essere preparati a migliaia o decine di migliaia di connessioni simultanee. I più antichi server che il web abbia visto negli anni '80 creavano un nuovo processo Unix per ogni connessione, secondo uno schema familiare ai

programmatore: l'uso di `fork()` e `exec()` per avviare flussi di esecuzione separati al ricevimento di una richiesta. Quello schema è andato in crisi fin dagli anni '90 per la pesantezza del meccanismo sottostante nel sistema.

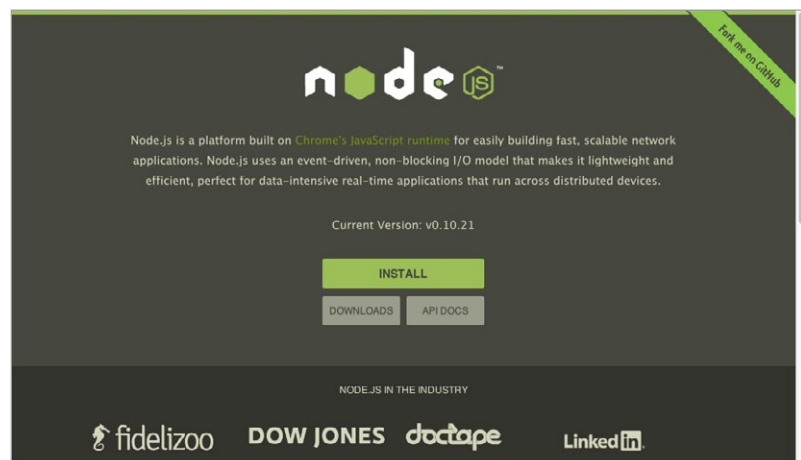
A ogni `fork`, il kernel deve creare una nuova mappatura di memoria virtuale e tutte le strutture interne che servono per gestire un processo in memoria. È un'operazione pesante, che non può essere interrotta e ripresa e deve essere adeguatamente protetta bloccando eventuali richieste al sistema che potrebbero andare in conflitto.

Fin dai primi tempi si sono cercate soluzioni migliori, basate sull'uso di `thread`. Un `thread` è un flusso di esecuzione concorrente all'interno di un processo. Non ha bisogno di una mappa di memoria che lo isoli dalla memoria degli altri processi, ma solo di uno stack privato per la storia della sua esecuzione. La creazione di un `thread`, quindi è un meccanismo molto veloce, ma anche uno stack per ogni `thread` ha il suo costo in memoria. Nel giro di pochi

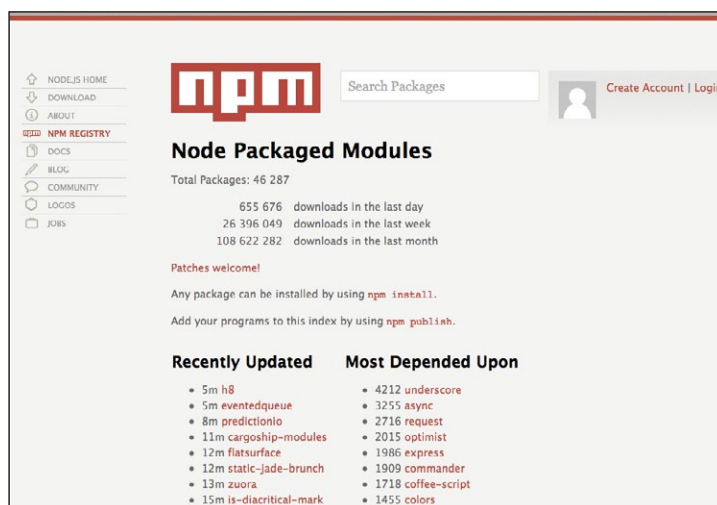


Il sito mobile di LinkedIn è stato sviluppato con node.js

anni, i server web più popolari (Apache e IIS) hanno adottato un modello basato sulla preallocazione di processi e di `thread` all'interno di ogni processo e sulla condivisione di connessioni da parte di ogni `thread` per limitare la creazione di ulteriori `thread` migliorando di parecchio le prestazioni.

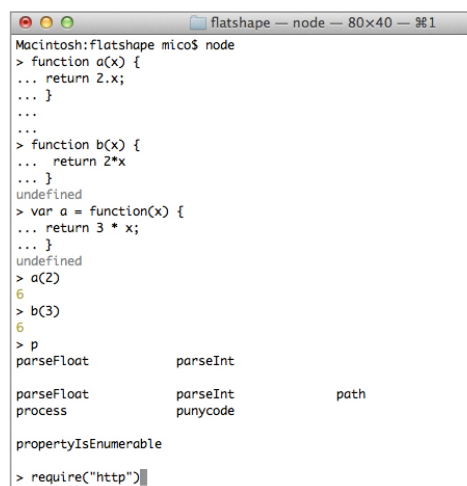


La home page di node.js



La pagina che organizza tutti i moduli che estendono node.js

Una sessione interattiva con node.js dalla riga di comando. Stiamo sperimentando con le funzioni.



In ultima analisi, l'ottimizzazione mirata di un server richiede un'analisi profonda del sistema, un compromesso basato sulla memoria e sul numero di Cpu a disposizione, in sostanza, somiglia a sufficienza alla magia nera.

Chi volesse approfondire la storia delle risposte architetturali al problema della disponibilità di un server può partire dal c10k challenge, la sfida di 10.000 connessioni concorrenti per server (www.kegel.com/c10k.html), che nasce alla fine degli anni '90, dopo che la prodezza di servire questo numero di connessioni era stata compiuta da cdrom.com. Più recentemente si è alzata di parecchio l'asticella, portando l'obiettivo a dieci milioni di connessioni. Si veda un articolo su highscalability.com (bit.ly/C10Mchallenge). Secondo l'autore, dieci milioni di connessioni simultanee richiedono innovazioni architetturali ancora più radicali, perché il kernel viene spinto ai suoi limiti e conviene portare parte della gestione del networking all'interno del server. Si consideri anche che

il port number, il numero che identifica una connessione socket, è un signed short, e questo impone un limite di 65.536 connessioni per indirizzo. Per fortuna, una connessione è identificata anche da altri codici, per esempio l'indirizzo remoto da cui è partita la connessione.

Insomma, questo genere di performance porta a sfide affascinanti, ma troppo complesse per tentarne una semplificazione in pochi paragrafi. Ci fermiamo qui ritenendo di avere dato un contorno preciso allo sfondo del problema dell'efficienza di un server e delle motivazioni che hanno portato alla nascita di node.js.

Un ambiente bello e funzionale

Nel 2009, alla JSConf, un meeting dedicato a JavaScript (jsconf.eu/2009/), insieme a esperti affermati, come Douglas Crockford e star come Thomas Fuchs, l'autore di Scriptaculous, e John Resig, l'autore di JQuery, si presentò un ragazzo, Ryan Dahl, che nel curriculum si definisce programmatore freelance.

Dahl, è un giovane programmatore sul quale c'è una curiosa mancanza di notizie (www.quora.com/Ryan-Dahl/Who-is-Ryan-Dahl). Il suo esordio sulla ribalta internazionale avviene, come spesso accade, sommando due più due in modo originale.

Da un lato Dahl sposa la filosofia architetturale del server web basato su un singolo thread di esecuzione e su routine di I/O non bloccanti. Un'architettura già stata sperimentata in diverse occasioni, per esempio il server Tornado, scritto in Python, che fa da motore a Friendfeed, un sito sociale caduto in disuso e comprato da Facebook. Anche il server Nginx, realizzato in C e rilasciato nel 2004 è un esempio di architettura basata su un singolo thread.

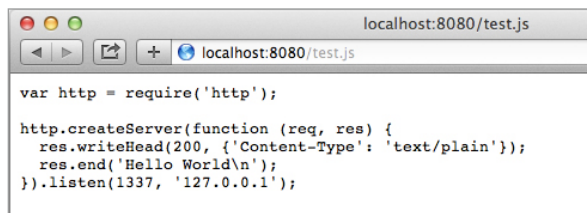
Il core di node.js è realizzato da Dahl in C++ cercando di creare un flusso di esecuzione senza interruzioni intorno alla lettura e scrittura sui socket delle connessioni.

Dall'altro lato, Dahl si pone il problema di cercare un linguaggio adatto per la programmazione applicativa, cioè il linguaggio utilizzato da chi programma applicazioni sul motore.

Lo schema del server impone la

In modalità interattiva carichiamo il modulo http e otteniamo una visualizzazione del suo contenuto



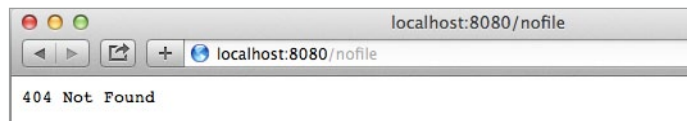


```
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
```

Il nostro semplice file server. Richiediamo un file esistente.

Questa volta abbiamo indicato una Uri non esistente.



```
404 Not Found
```

programmazione basata su eventi, secondo il pattern reactor (http://en.wikipedia.org/wiki/Reactor_pattern). Il server, in questa architettura, occupa un singolo thread e un singolo processo e tutti i compiti che richiedono elaborazione e attesa devono essere delegati a funzioni, che saranno eseguite quando è il momento. L'architettura, quindi, fa largo uso di callback.

Ne segue che un linguaggio adatto per questa piattaforma deve essere un linguaggio funzionale, ossia un linguaggio in cui le funzioni sono oggetti primitivi, come i numeri e le stringhe. Un altro requisito architetturale del linguaggio è di non ostacolare una soluzione single threaded non bloccante.

Dahl trovò una soluzione open source, che rispettava tutti i requisiti e con ottime doti di performance: la macchina virtuale V8, il motore JavaScript di Google Chrome.

Unendo il server C++ con l'interprete JavaScript, Dahl ottiene un ambiente operativo molto leggero (si vedano i sorgenti su github.com/joyent/node), che unisce a performance potenzialmente elevatissime, la semplicità di programmazione di JavaScript.

Mettiamoci le mani

Installare node.js è semplicissimo: basta scaricare dal sito nodejs.org il pacchetto appropriato per Windows o OS X e in poco tempo si può partire.

Per mettere alla prova il server, basta creare un file di prova, per esempio con questo testo:

```
var http = require('http');

http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

Dalla riga di comando lanciamo *node test.js* e abbiamo un server in esecuzione sulla nostra macchina. Aprendo la pagina localhost:1337 ci vedremo rispondere *Hello World*.

Con questo abbiamo terminato il rituale che ci si aspetta da qualunque introduzione a qualunque linguaggio o ambiente operativo.

Qualcosa di più concreto

Passiamo a esaminare un esempio di file server statico, in grado di inviare file statici: più o meno il minimo che ci si aspetta da un server web e, tutto sommato, buona parte di quello che hanno fatto i server web per tutti gli anni '80. Per il codice ricorriamo a un esempio popolare sul web, nato da un tutorial pubblicato su net.tutsplus.com col titolo *Learning Server-Side JavaScript with Node.js*. Noi useremo una versione aggiornata alla release corrente di node.js di questo tutorial, prendendo ispirazione da una versione aggiornata, che si trova su pastebin.com/vbxyKAa1.

Ecco il testo del programma:

```
var sys = require("sys"),
    http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs");

http.createServer(function(request, response) {
  var uri=url.parse(request.url).pathname;
  var filename = path.join(process.cwd(), uri);
  path.exists(filename, function(exists) {
    if(!exists) {
      response.writeHead(404, {"Content-Type": "text/plain"});
      response.write("404 Not Found\n");
      response.end();
      return;
    }
    fs.readFile(filename, "binary", function(err, file) {
      if(err) {
        response.writeHead(500, {"Content-Type": "text/plain"});
        response.write(err + "\n");
        response.end();
        return;
      }
      response.writeHead(200);
      response.write(file, "binary");
      response.end();
    });
  }).listen(8080);

sys.puts("Server running at http://localhost:8080/");
```

```
fs.readFile(filename, "binary", function(err, file) {
  if(err) {
    response.writeHead(500, {"Content-Type": "text/plain"});
    response.write(err + "\n");
    response.end();
    return;
  }
  response.writeHead(200);
  response.write(file, "binary");
  response.end();
});
}).listen(8080);

sys.puts("Server running at http://localhost:8080/");
```

Esaminiamo questo codice passo per passo.

I moduli di node.js

Le prime righe, che iniziano con *require*, fanno riferimento a una soluzione per la modularizzazione di JavaScript, le cui specifiche si trovano su commonjs.org (wiki.commonjs.org/wiki/Modules).

Uno standard per la creazione e la pubblicazione di moduli è vitale per la creazione di un ambiente operativo sofisticato. Vedremo in maggiore dettaglio questa organizzazione in un prossimo articolo. Il comando *require*, che è un'aggiunta a JavaScript e, quindi, non funzionerebbe in un browser, carica alcuni moduli che provvedono le funzionalità che il nome lascia pensare. I moduli *sys*, *http*, *url*, *path* e *fs* offrono rispettivamente primitive per l'accesso al sistema, per la gestione del protocollo *http*, per il parsing di *Url*, per la gestione di *path* e del file system locale.

Per sperimentare cosa contiene un modulo, anche in assenza di documentazione, o della voglia di andarla a cercare sul

web, si può usare direttamente `node.js`. Digittiamo da riga di comando

```
node
```

Lanciato senza un nome di file, `node.js` passa in modo interattivo, leggendo i comandi dal terminale. Al prompt di `node`, diamo il comando

```
require("fs")
```

Node risponderà caricando il modulo richiesto e esponendo il valore di ritorno del caricamento, che è l'oggetto stesso. Come ci si può aspettare, se si è fatto un po' di debugging JavaScript, se una funzione restituisce un oggetto, l'interprete interattivo lo stampa sul video in una rappresentazione testuale, quindi vedremo comparire una versione abbreviata del testo del codice del modulo:

```
> fs
{ Stats: [Function],
  exists: [Function],
  existsSync: [Function],
  readFile: [Function],
  readFileSync: [Function],
  close: [Function],
  closeSync: [Function],
  open: [Function],
  .....
  FileWriteStream:
    { [Function: WriteStream]
      super_:
        { [Function: Writable]
          WritableState: [Function:
            WritableState],
          super_: [Object] },
      SyncWriteStream:
        { [Function: SyncWriteStream]
          super_:
            { [Function: Stream]
              super_: [Object],
              Readable: [Object],
              Writable: [Object],
              Duplex: [Object],
              Transform: [Object],
              PassThrough: [Object],
              Stream: [Circular] } } }
```

Da qui vediamo che il modulo definisce

delle funzioni, per esempio `Stats`, `exists`, `existsSync`, e delle strutture dati più complesse, come `FileWriteStream`.

Se proviamo a digitare `fs` al prompt di `node`, otterremo di nuovo il codice che ha seguito `import`. Proviamo a scrivere `fs.File` e premere un tab, `node` proporrà il completamento automatico e mostrerà i due oggetti definiti nel modulo `fs`, che iniziano con `File`.

Con questo sistema possiamo farci un'idea rapida del contenuto di un modulo. Tutti i moduli disponibili sono documentati su npmjs.org, il sito ufficiale dei moduli `node.js`.

Come operano le funzioni

Subito dopo l'importazione dei moduli, abbiamo la chiamata alla funzione

```
http.createServer(...).listen(8080)
```

La variabile `http` contiene, come abbiamo visto, metodi e oggetti del modulo `http`. La funzione `createServer` viene chiamata passando come parametro una funzione, la cui definizione prosegue su più righe, fino alla parentesi tonda chiusa prima di `listen(8080)`. La chiamata a `listen` è eseguita dall'oggetto creato da `createServer`, cioè dal server.

In una riga abbiamo due stilemi familiari a chi usa JQuery: la chiamata di funzioni a catena e la definizione di una funzione anonima fra le parentesi di una chiamata, una delle caratteristiche di JavaScript, che abbiamo visto di recente filtrare in altri linguaggi.

Facciamo un breve ripasso sulle funzioni in JavaScript usando `node.js` interattivamente.

Proviamo, ad avviare `node` senza parametri e a digitare il codice seguente quando appare un prompt:

```
function b(x) {
  return 2*x
}
```

Se scriviamo `b` e premiamo invio, `node.js` ci conferma che si tratta del nome di una funzione. Invochiamola

```
b(3)
```

La risposta sarà ovviamente 6.

Questo stile della dichiarazione di funzioni è comune a molti altri linguaggi e non ha nulla di nuovo. Adesso proviamo a digitare il codice seguente:

```
var a = function() {
  return 3;
}
```

Stiamo assegnando alla variabile `a` una funzione. Proviamo a invocare questa funzione, digitando `a()` e otterremo la risposta, cioè 3.

Questo esempio ci mostra un aspetto interessante del linguaggio: i nomi di funzioni sono pari ai nomi di variabile, quindi non sono in una tabella separata dai nomi di variabile. Allo stesso modo, un nome di variabile è capace di contenere un numero, una stringa o una funzione, questo fa di JavaScript un linguaggio funzionale e giustifica l'affermazione di Crockford, che JavaScript è un Lisp con gli abiti del C. Nel nostro codice usiamo un terzo approccio: approfittiamo della possibilità di creare una funzione anonima creando sul posto, cioè fra le parentesi tonde della chiamata, la funzione da passare come parametro a `http.createServer`. Troviamo questo modo di creare funzioni a ogni passo nel codice delle librerie JavaScript più famose.

Il server vero e proprio

La funzione che gestisce le richieste in arrivo, il parametro di `createServer`, riceve due parametri: un oggetto che descrive la richiesta e uno che descrive la risposta. Si tratta di due strutture dati che dipendono soprattutto dal protocollo `http`, sono molto simili in qualsiasi linguaggio e sono presenti in ogni ambiente di programmazione di applicazioni web.

La richiesta contiene dettagli come l'indirizzo remoto del client, la Url che è stata passata e diverse altre informazioni legate al protocollo, come i cookie. La risposta contiene gli header `http` e il testo della pagina.

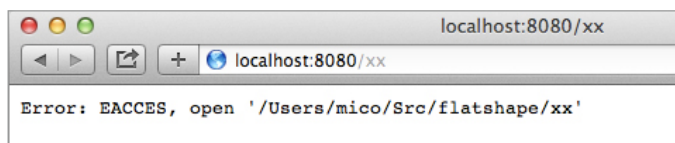
Vediamo come ci comportiamo per il nostro server statico.

La prima riga del codice

```
var uri = url.parse(request.url).
pathname;
```

estrae dalla *Url* la *Uri* richiesta utilizzando la funzione `parse` dell'oggetto `url`.

Abbiamo chiesto
un file non
leggibile.



La *Uri* è la porzione della *Url* che indica il file richiesto. Per esempio, se la *Url* è *localhost:8080/path/del/miofile.html*, stiamo parlando del file *path/del/miofile.html*. La seconda riga del codice

```
var filename=path.join(process.  
cwd(), uri);
```

usa *path.join* per unire insieme, con il separatore adatto per il sistema operativo in uso, la directory in cui è in esecuzione il server (*process.cwd*) con la *Uri* del file richiesto. Questo è un passaggio indispensabile soprattutto da quando Microsoft ha scelto come separatore uno slash girato per l'altro verso, considerando che la quota di sistemi operativi per server non Microsoft e non basati su unix è un errore di arrotondamento, anzi, calcolando telefoni e tablet basati su iOS e Android, il numero di sistemi unix nel mondo farebbe girare la testa a Thompson e Ritchie. Un server migliore userebbe un parametro di configurazione per la home dei documenti, che di sicuro non sarebbe la cartella che contiene il codice dell'applicazione, ma questo è un dettaglio facilmente migliorabile. Il punto successivo è la verifica dell'esistenza del file, con *path.exists*.

```
path.exists(filename,   
function(exists) [...])
```

Anche in questo caso c'è una funzione anonima. La realizzazione di *path.exists*, infatti, invece di limitarsi a rispondere sì o no, passa il valore di verità (il parametro che chiamiamo *exists*) a una funzione che lo usa.

Il resto del codice è abbastanza semplice, il primo *if*, *if (!exists)* gestisce il caso in cui il file non è stato trovato. Se non troviamo il file bisogna restituire un errore 404, cioè una pagina con 404 negli header

```
response.writeHead(404,  
{ "Content-Type": "text/plain" });
```

la funzione *writeHead* dell'oggetto *response*, che ci viene passato dalla creazione del server in alto, scrive sul socket di connessione un'intestazione con il codice 404 e gli header, che sono passati nel secondo parametro come array di stringhe, quindi fra graffe.

```
response.write("404 Not  
Found\n");  
response.end();
```

```
simpleFileServer.js  
  
var sys = require("sys"),  
    http = require("http"),  
    url = require("url"),  
    path = require("path"),  
    fs = require("fs");  
  
http.createServer(function(request, response) {  
    var uri = url.parse(request.url).pathname;  
    var filename = path.join(process.cwd(), uri);  
    path.exists(filename, function(exists) {  
        if(!exists) {  
            response.writeHead(404, { "Content-Type": "text/plain" });  
            response.write("404 Not Found\n");  
            response.end();  
            return;  
        }  
  
        fs.readFile(filename, "binary", function(err, file) {  
            if(err) {  
                response.writeHead(500, { "Content-Type": "text/plain" });  
                response.write(err + "\n");  
                response.end();  
                return;  
            }  
            response.writeHead(200);  
            response.write(file, "binary");  
            response.end();  
        });  
    }).listen(8080);  
  
sys.puts("Server running at http://localhost:8080/");
```

Il codice del nostro minimo server statico aperto nell'editor Emacs.

Il contenuto della pagina viene impostato a un laconico *404 Not Found* e la risposta viene chiusa (*response.end*), completando l'invio dei dati al client e liberando la connessione.

Il caso in cui il file esiste offre due possibilità: la lettura del file e l'invio al browser può avvenire senza errori, oppure qualcosa va storto. Simulare un errore è facile, se si lavora su Linux o su un Mac basta creare un file e togliere a chiunque il permesso di leggerlo, per esempio

```
touch xx  
chmod 000 xx
```

un accesso a *localhost:8080/xx* ci rimanda l'errore 500 (*Internal Server Error*) che ci aspettiamo. Il corpo della pagina contiene il messaggio di errore che abbiamo ricevuto, in questo caso *Error: EACCES, open '/Users/mico/Src/nodeTest/xx'*.

Infine, se tutto va bene impostiamo lo header a 200 e, nel corpo della risposta, inviamo il file con *response.write(file, "binary")*.

Per arrivare a replicare le funzioni di base di un server apache ci manca la gestione di qualche parametro di configurazione, dell'accesso con password a certe directory e di un insieme di tipi di file più ricco di *text/plain*, ma come inizio non è male, soprattutto per 31 righe di codice.

Conclusioni

Node.js è una piattaforma software moderna, adatta per server destinati a servire molte connessioni contemporanee, anche con hardware limitato. Le prestazioni sono eccellenti, considerando che

il JavaScript è uno strato sottile intorno a un server in C++, e le funzioni principali sono realizzate in codice nativo e richiamate da un wrapper JavaScript.

La macchina virtuale V8, su cui si basa node.js è una delle più veloci in circolazione e dispone di un compilatore *just in time (jit)* continuamente ottimizzato. Le prestazioni che si possono ottenere, quindi sono in generale migliori di quelle di Python, come si può verificare nei diversi benchmark sul *Language Shootout* (benchmarksgame.alioth.debian.org).

Il vantaggio più rilevante della piattaforma è che si può usare lo stesso linguaggio familiare agli sviluppatori lato client anche sul server, portandosi dietro anche l'attitudine a demandare a funzioni callback tutto quello che può bloccare il flusso di elaborazione principale. La portabilità è ottima, perché node.js gira su unix e su Windows, oltre a essere parte dell'offerta standard del cloud di Amazon e di Azure, il cloud di Microsoft.

Il corredo di moduli a disposizione è ampio e l'installazione dei moduli è semplice e gestita da un package manager simile a quello degli ambienti Linux. Nel giro di pochi minuti si può creare un ambiente di pubblicazione sul cloud, per esempio Azure, usando un repository Git e un server locale per gestire lo sviluppo, pubblicando in remoto con un push di Git. Lo sviluppo locale è semplice sia su Windows, sia su Linux o OS X e gli strumenti a disposizione sono notevoli. Dal lato negativo dobbiamo solo registrare che occorre familiarizzarsi con un ambiente nuovo e aderire a un modello in cui nel flusso di elaborazione principale non deve avvenire nulla che possa bloccare l'intero server. •