LA PRIMA RIVISTA ITALIANA DEDICATA A LINUX E ALL'OPENSOURCE

www.oltrelinux.com

Linux&C.

MENSILE - ANNO 12 NUMERO 74

€ 3,50

CONFRONTO

VIRTUALIZZAZIONE

KVM, XEN e VirtualBOX

ARCHITETTURA, INTERNALS, INTERFACCE GRAFICHE E A LINEA DI COMANDO

LE SOLUZIONI OPENSOURCE PIU' IMPORTANTI

> SERVER/APPLICATION CONSOLIDATION
> CLOUD COMPUTING > LIVE MIGRATION

ISSN 1129-2296

Android: abbiamo provato Honeycomb e Nvidia TEGRA!
PROVATI PER VOI: TABLET ACER ICONIA A500 E SMARTPHONE LG OPTIMUS DUAL

La videoconferenza facile come premere un tasto: BigBlueButton
GESTIRE CONFERENZE, AUDIO E VIDEO, PUO' ESSERE COMPLESSO: MA NON CERTO CON BBB!

Firewall: come gestire Iptables tramite template grazie a FERM
CHIUNQUE ABBIA USATO IPTABLES NE CONOSCE I LIMITI DI USABILITA': FERM E' UNA SOLUZIONE

Drupal: creare view e temi personalizzati per il nostro sito
LE ESIGENZE DI UN SITO POSSONO ESSERE MOLTO PARTICOLARI: MA DRUPAL PUO' TUTTO!

E non finisce qua...
- KERNEL 3.0: LINUS HA ANNUNCIATO LO STORICO PASSAGGIO
- MIGRIAMO LA RETE LOCALE A IPV6 PER ESSERE PRONTI
- NETWORK TIMING: SINCRONIZZARSI AL MICROSECONDO
- HADOOP: COME SVILUPPARE APPLICAZIONI MAP/REDUCE

Spedizione in A.P. - 45% Art. 2 comma 20/b legge 662/96 - Filiale di Milano Linux&C. - Anno 12 Num. 74 LUG/AGO 2011 - DISTRIBUZIONE ME.PE. - PISCOPO EDITORE S.R.L.

Programmare **Apache Hadoop** con **Python**

Hadoop è il framework Map/Reduce Open Source più diffuso: vediamo come sfruttarlo anche con Python, riducendo i tempi di sviluppo

Simone Leo
s.leo@oltrelinux.com

Simone Leo ha conseguito la laurea in Ingegneria Elettronica presso l'Università degli Studi di Cagliari nel 2002. Dopo un periodo di lavoro nel settore industriale come amministratore di sistemi, è entrato a far parte nel 2006 del gruppo Distributed Computing del CRS4, inizialmente come Software Engineer e, dal 2008, come Ricercatore. Le sue attività di ricerca si concentrano principalmente su modelli innovativi di computazione distribuita, con particolare enfasi sull'applicazione degli stessi alla biologia computazionale.

Nell'articolo precedente abbiamo fatto conoscenza con Hadoop, l'implementazione open source in Java di MapReduce, modello per l'analisi distribuita di dati su larga scala originariamente sviluppato da Google ed abbiamo visto come installare Hadoop, configurarlo ed eseguire word count, l'equivalente MapReduce di *"hello world"*.

Obiettivo di questa seconda parte è lo sviluppo di applicazioni che sfruttino un cluster Hadoop; in particolare, vedremo come sia possibile svilupparle in Python, un linguaggio che gode di notevole popolarità, principalmente grazie alla rapidità di sviluppo, alla quantità di moduli disponibili ed alla relativa facilità di estensione.

Anche se, come accennato nella prima parte, esistono implementazioni alternative di MapReduce che supportano Python direttamente, Hadoop è senza dubbio quella più diffusa: al momento in cui scrivo, le organizzazioni che usano Hadoop in produzione o a scopo didattico sono oltre 120. Oltre a Yahoo!, che fa la parte del leone con oltre 36000 macchine, tra i maggiori utilizzatori ci sono Facebook, LinkedIn, EBay e Twitter. Tuttavia, il matrimonio tra Python e Hadoop non è semplice: come vedremo, le opzioni integrate in Hadoop hanno diversi inconvenienti che hanno spinto l'autore di questo articolo ed il suo boss a sviluppare Pydoop, un'API Python basata su wrapper Boost per l'interfaccia C/C++ di Hadoop.

Programmazione MapReduce in Hadoop

Come abbiamo visto nella puntata precedente, Hadoop si divide in due componenti fondamentali: un filesystem distribuito (HDFS) ed il framework di calcolo MapReduce. L'esempio tipico di programmazione MapReduce è word count, un'applicazione che conta l'occorrenza di ciascuna parola in un insieme di file di testo: il mapper emette una coppia (`parola, 1`) per ogni parola in ingresso, il framework raggruppa automaticamente le coppie per parola ed il reducer somma tutti i valori corrispondenti a ciascuna parola. Nel listato 1 riportiamo l'implementazione Java di word count inclusa negli esempi di Hadoop.

Il punto di forza di MapReduce sta nel fatto che il programmatore deve scrivere, in genere, soltanto il codice che definisce le trasformazioni operate da `map` e `reduce`: la distribuzione del lavoro sui vari nodi è gestita da Hadoop, come anche il riordinamento (*shuffle & sort*) delle coppie chiave/valore emesse dal mapper prima dell'invio al reducer (tutt'altro che banale, visto che i vari mapper e reducer girano su macchine diverse).

Dall'esempio si capisce subito che l'utilizzo dell'API MapReduce di Hadoop non è immediato: per un'applicazione così semplice, le righe di codice non sono poche (contando anche gli `import` all'inizio ed il `main` alla fine, qui omessi per brevità, il programma è di 70 righe). Inoltre, molte organizzazioni dispongono di un notevole patrimonio di software preesistente, non necessariamente in Java, del quale è importante massimizzare il riuso. Fortunatamente, esistono diverse soluzioni per lo sviluppo in linguaggi diversi da Java, alcune delle quali incluse nella distribuzione ufficiale di Hadoop: Streaming, Jython e Pipes.

L1 - WordCount in Java

```
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new
StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
  public static class IntSumReducer
       extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

L2 - Mapper per Streaming

```
#!/usr/bin/env python
import sys
for line in sys.stdin:
  for word in line.split():
    print '%s\t1' % word
```

L3 - Reducer per Streaming

```
#!/usr/bin/env python
import sys

def serialize(key, value):
  return '%s\t%d' % (key, value)

def deserialize(line):
  key, value = line.split("\t", 1)
  return key, int(value)

def main():
  prev_key, out_value = None, 0
  for line in sys.stdin:
    key, value = deserialize(line)
    if key != prev_key:
      if prev_key is not None:
        print serialize(prev_key, out_value)
        out_value = 0
      prev_key = key
    out_value += value
  print serialize(key, out_value)

if __name__ == "__main__":
  main()
```

Streaming

Streaming è un tool (implementato come libreria Java) che consente di utilizzare un eseguibile esterno come mapper e/o reducer: le coppie chiave/valore, opportunamente serializzate, vengono scambiate con il framework via standard input/output. Il vantaggio di questa soluzione consiste nella possibilità di programmare in qualsiasi linguaggio: mapper e reducer sono infatti visti dal framework come processi *"black box"* con cui interagire solo tramite i flussi di I/O. Streaming ha, però, diversi svantaggi: ad esempio non è possibile scrivere componenti personalizzati al di fuori del mapper e del reducer (record reader, partitioner, etc.), fino alla versione 0.20.2 (l'ultima stabile) il protocollo di comunicazione è solo testuale e la gestione del flusso di coppie chiave/valore è più macchinosa rispetto all'API Java, soprattutto nel reducer.

Per illustrare al meglio quest'ultimo aspetto, vediamo un'implementazione di *word count* per Streaming che utilizza due script Python come mapper (listato 2) e reducer (listato 3). Il mapper è piuttosto semplice: il framework alimenta lo script con un flusso di linee di testo e si aspetta di ricevere le coppie chiave/valore intermedie ancora come linee di testo, con la chiave e il valore separati da un carattere di tabulazione. Le coppie intermedie vengono deserializzate da Streaming, ordinate per chiave da Hadoop, riserializzate e inviate al reducer, dove il controllo passa di nuovo al nostro script, questa volta per il *reduce*.

Vale la pena di soffermarsi sulle differenze tra lo sviluppo con l'API Java e Streaming. Nel primo caso si implementa un metodo `reduce` che il framework chiama una volta per ogni chiave intermedia, passandogli la chiave stessa ed un iteratore su tutti i valori ad essa corrispondenti: tutto ciò che resta da fare è sommare tra loro gli elementi forniti dall'iteratore. Nel secondo caso si scrive un programma completo, che riceve da standard input un flusso di coppie chiave/valore separate da tabulazione ed ordinate per chiave: la deserializzazione dell'input, l'accumulazione dei valori corrispondenti a ciascuna chiave e la serializzazione dell'output sono tutte operazioni a carico dello sviluppatore. Inoltre, i passaggi di serializzazione e deserializzazione aumentano il carico di lavoro per l'applicazione, che girerà più lentamente rispetto alla controparte in Java.

Per lanciare questo esempio, salvate gli script come `mapper.py` e `reducer.py` ed eseguite il seguente comando:

```
$ hadoop jar ${HADOOP_HOME}/contrib/streaming/
                    hadoop-0.20.2-streaming.jar -mapper mapper.py \
  -reducer reducer.py -input wc_input -output wc_output \
  -file mapper.py -file reducer.py
```

(`wc_input` è la directory HDFS che contiene il testo da analizzare).

Jython

Jython è un'implementazione in Java del linguaggio Python. Tra i vantaggi che offre c'è, soprattutto, quello di consentire l'utilizzo di librerie Java, comprese quelle di Hadoop, ma ha lo svantaggio, rispetto all'implementazione ufficiale (CPython), di essere sempre indietro di una o più release, di disporre di una standard library incompleta e di non poter utilizzare package Python di terze parti che includano estensioni in C/C++ (tra cui compaiono molte di quelle fondamentali per il calcolo scientifico). Con Jython l'accesso alle feature di Hadoop è completo, ma la programmazione soffre di limitazioni piuttosto restrittive.

Nel listato 4 riportiamo l'implementazione Jython di word count, tratta dagli esempi di Hadoop.

L4 - WordCount in Jython

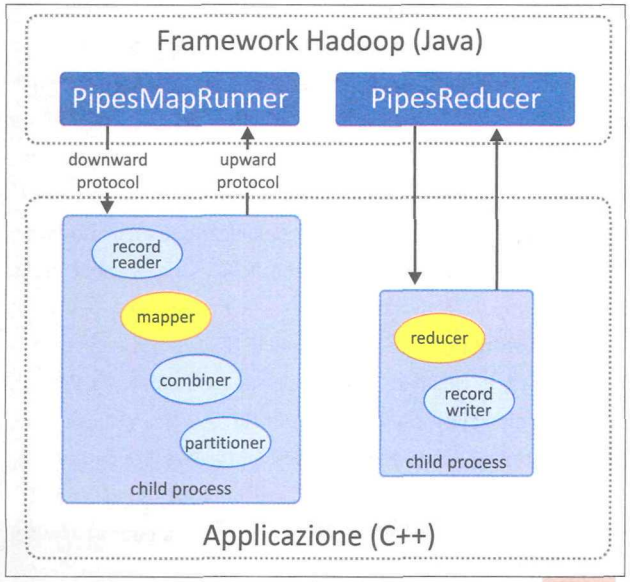
```
class WordCountMap(Mapper, MapReduceBase):
  one = IntWritable(1)
  def map(self, key, value, output, reporter):
    for w in value.toString().split():
      output.collect(Text(w), self.one)

class Summer(Reducer, MapReduceBase):
  def reduce(self, key, values, output, reporter):
    sum = 0
    while values.hasNext():
      sum += values.next().get()
    output.collect(key, IntWritable(sum))
```

Pipes

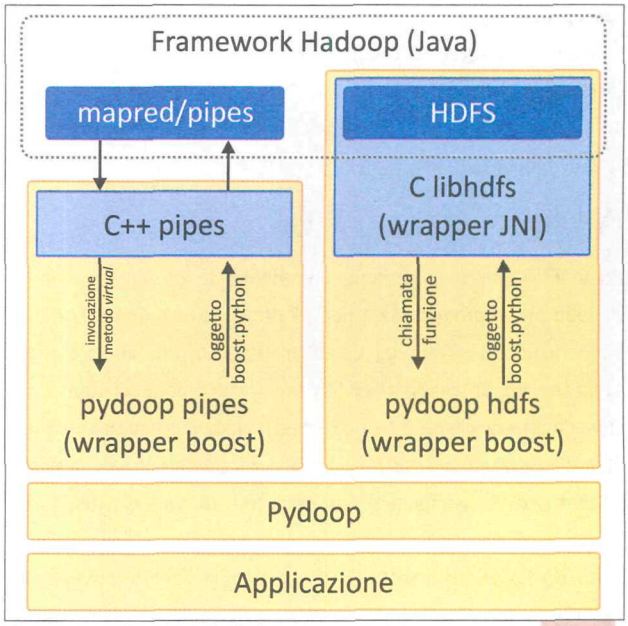
Pipes è un modulo che fornisce un'API MapReduce in C++ per Hadoop. Il funzionamento di Pipes è schematizzato in figura 1: la parte Java (integrata in Hadoop) scambia oggetti serializzati tramite un socket persistente con la parte C++ (dove risiede l'applicazione) secondo uno specifico protocollo binario.

La linea di demarcazione tra Java e C++ varia a seconda di quale dei due linguaggi implementa il *record reader* ed il *record writer*, il che dipende dai valori di `hadoop.pipes.java.recordreader` e `hadoop.pipes.java.recordwriter`: se entrambi sono impostati a `true`, i record di input sono creati lato Java e passati al *mapper* in C++ tramite il downward protocol (da Java verso C++); le coppie chiave/valore intermedie emesse dal *mapper* tornano sul lato Java attraverso l'upward protocol (da C++ verso Java) e, dopo il raggruppamento per chiave, passano al *reducer* in C++, ancora tramite il downward protocol; infine, le coppie chiave/valore emesse dal *reducer* passano al *record writer* lato Java, ancora tramite l'upward protocol. Se `hadoop.pipes.java.recordreader` è `false`, il framework invia gli *input split* (descrizioni logiche dei sottoinsiemi in cui i dati sono suddivisi) al record reader in C++, che legge i dati da HDFS e



Schema di funzionamento di Pipes. Il framework comunica con l'applicazione tramite un socket dedicato

1



Integrazione di Pydoop in Hadoop

2

R1 - Installare ed utilizzare Pydoop

I prerequisiti minimi di Pydoop sono Python 2.6, Apache Hadoop 0.20.2 e Boost.Python 1.40; abbiamo visto come installare e configurare Hadoop nel numero scorso, mentre Python è preinstallato in tutte le maggiori distribuzioni e Boost è quasi sempre disponibile come package opzionale. In alcune distribuzioni, le librerie Boost sono suddivise in diversi sub-package: in questo caso, per soddisfare i prerequisiti di Pydoop, è sufficiente installare la sola libreria Python. Su Ubuntu, per esempio, eseguite:

```
$ sudo apt-get install libboost-python-dev
```

Terminata l'installazione dei prerequisiti, scaricate Pydoop da http://sourceforge.net/projects/pydoop/files/, aprite l'archivio e spostatevi nella directory così creata. Infine, eseguite:

```
$ export JAVA_HOME=<LA_VOSTRA_JAVA_HOME>
$ export HADOOP_HOME=<LA_VOSTRA_HADOOP_HOME>
$ python setup.py build
$ sudo python setup.py install --skip-build
```

Le variabili d'ambiente **JAVA_HOME** e **HADOOP_HOME** comunicano a Pydoop le directory di installazione di Java (JDK) e Hadoop; i rispettivi default sono **/opt/sun-jdk** e **/opt/hadoop**.
Per eseguire le applicazioni, si procede come in Pipes:

```
${HADOOP_HOME}/bin/hadoop fs -put input{,}
${HADOOP_HOME}/bin/hadoop fs -put wordcount.py{,}
${HADOOP_HOME}/bin/hadoop pipes -conf conf.xml -input input
-output output
```

Il file di configurazione del job, qui chiamato **conf.xml**, ha la stessa struttura dei file di configurazione di Hadoop che abbiamo incontrato nell'articolo precedente. La configurazione minima è la seguente:

```
<?xml version="1.0"?>
<configuration>
<property>
  <name>hadoop.pipes.executable</name>
  <value>wordcount.py</value>
</property>
<property>
  <name>hadoop.pipes.java.recordreader</name>
  <value>true</value>
</property>
<property>
  <name>hadoop.pipes.java.recordwriter</name>
  <value>true</value>
</property>
</configuration>
```

Il valore di **hadoop.pipes.executable** è il percorso HDFS (in questo caso relativo alla home) dell'eseguibile che lancia l'applicazione; gli altri due parametri vanno impostati a **false** se la vostra applicazione implementa un record reader e/o un record writer. Nel file di configurazione è possibile impostare parametri standard di Hadoop (per es., **mapred.map.tasks**) o specifici della vostra applicazione.

crea i record di input per il *mapper*; analogamente, se **hadoop.pipes.java.recordwriter** è **false**, sarà un *record writer* in C++ a scrivere i record di output su HDFS.
Oltre a Pipes, il programmatore C/C++ ha a disposizione **libhdfs**, un'API C per HDFS. A differenza di Pipes, questa è implementata come wrapper JNI.

Pydoop

Come mostrato nella sezione precedente, le opzioni built-in di Hadoop per la programmazione in Python – Streaming e Jython – hanno vantaggi e svantaggi complementari. Un'alternativa è offerta da Dumbo, un wrapper per Streaming che espone all'utente una comoda interfaccia object-oriented ed include meccanismi per facilitare l'esecuzione dei job. Purtroppo, però, Dumbo non dà la possibilità di scrivere componenti opzionali (record reader/writer, partitioner) né di accedere ad HDFS.
Tutto ciò ha spinto il sottoscritto e Gianluigi Zanetti a sviluppare Pydoop, un'API MapReduce e HDFS per Hadoop, a supporto delle attività di ricerca del gruppo Distributed Computing del CRS4.

Il cuore di Pydoop è costituito da due estensioni implementate come wrapper Boost.Python per Pipes e libhdfs. L'architettura è mostrata in figura 2: nel caso di Pipes, il framework Java istanzia i componenti MapReduce (classi definite dall'utente che estendono le classi base di Pydoop, a loro volta dei wrapper per le classi C++) utilizzando una factory fornita dall'applicazione, ne chiama i metodi all'occorrenza (per es. **map**, a cui passa una coppia chiave/valore di input) e riceve gli oggetti da essi restituiti (proseguendo con l'esempio di **map**, la coppia chiave/valore intermedia); la comunicazione tra il lato Java e quello C++ avviene tramite il protocollo di serializzazione sopra citato, mentre le conversioni tra C++ e Python sono gestite in maniera trasparente da Boost; nel modulo HDFS, il flusso di controllo è opposto: le chiamate di funzione sono effettuate da Pydoop, che riceve gli oggetti risultanti dall'API HDFS Java; i wrapper JNI e Boost effettuano le necessarie conversioni tra Java, C e Python.
Rispetto a Streaming, i vantaggi principali di Pydoop sono l'interfaccia object-oriented simile a quella Java, l'accesso a buona parte dei componenti MapReduce opzionali (*combiner, record reader, record writer* e *partitioner*) e l'API HDFS; rispetto a Jython, Pydoop ha il

vantaggio di consentire l'uso di package CPython. Nel seguito faremo riferimento alla versione 0.4.0 di Pydoop, la più recente al momento in cui scrivo: l'installazione del package e l'esecuzione delle applicazioni sono descritte nel riquadro 1; inoltre, tutto il codice che segue, insieme ad alcuni script per il lancio ed il test, è disponibile nel tarball sotto `examples/wordcount`.

Il listato 5 mostra l'implementazione di word count in Pydoop.

In Pipes, e quindi anche in Pydoop, tutta la comunicazione con il framework è mediata dall'oggetto context. Tramite il `context`, i *mapper* ricevono le coppie chiave/valore di input (in questo caso la chiave, pari all'offset in byte del file di input, non è usata) ed emettono le coppie chiave/valore intermedie; dal `context` tutti i componenti ricevono i parametri di configurazione, aggiornano i contatori ed inviano rapporti di stato. I contatori sono variabili definibili dall'utente che tengono traccia di un qualche parametro rilevante per l'applicazione; i rapporti di stato sono messaggi di testo arbitrari inviati al framework, particolarmente utili nei casi in cui l'analisi di un singolo record richieda un tempo considerevole, dato che Hadoop termina forzatamente i task che non leggono input, non scrivono output e non inviano rapporti di stato per un intervallo di tempo configurabile (il default è dieci minuti). I messaggi di stato ed i contatori sono visualizzati dall'interfaccia web di MapReduce; i valori finali dei contatori sono riportati anche nel job log e su standard output.

L5 - WordCount per Pydoop

```
#!/usr/bin/env python
import pydoop.pipes as pp

class Mapper(pp.Mapper):
  def map(self, context):
    words = context.getInputValue().split()
    for w in words:
      context.emit(w, "1")

class Reducer(pp.Reducer):
  def reduce(self, context):
    s = 0
    while context.nextValue():
      s += int(context.getInputValue())
    context.emit(context.getInputKey(), str(s))

if __name__ == "__main__":
  pp.runTask(pp.Factory(Mapper, Reducer))
```

L6 - Uso dei counter in Pydoop

```
class Mapper(pp.Mapper):
  def __init__(self, context):
    super(Mapper, self).__init__(context)
    context.setStatus("initializing mapper")
    self.inputWords = context.getCounter("WORDCOUNT",
                                         "INPUT_WORDS")

  def map(self, context):
    words = context.getInputValue().split()
    for w in words:
      context.emit(w, "1")
    context.incrementCounter(self.inputWords, len(words))
```

	Counter	Map	Reduce	Total
WORDCOUNT	OUTPUT_WORDS	7,318	6,014	13,332
	INPUT_WORDS	29,459	0	29,459
Job Counters	Launched reduce tasks	0	0	2
	Launched map tasks	0	0	2
	Data-local map tasks	0	0	2
FileSystemCounters	FILE_BYTES_READ	0	84,210	84,210
	FILE_BYTES_WRITTEN	84,334	84,210	168,544
Map-Reduce Framework	Reduce input groups	0	6,014	6,014
	Combine output records	0	0	0
	Map input records	2	0	2
	Reduce shuffle bytes	0	63,876	63,876
	Reduce output records	0	0	0
	Spilled Records	7,318	7,318	14,636
	Map output bytes	69,562	0	69,562
	Map input bytes	0	0	0

La tabella che contiene i counter nell'interfaccia web del Job Tracker di Hadoop

3

L7 - Record Reader in Pydoop

```
import struct
import pydoop.hdfs as hdfs

class Reader(pp.RecordReader):
  def __init__(self, context):
    super(Reader, self).__init__(context)
    self.isplit = pp.InputSplit(context.getInputSplit())
    self.file = hdfs.open(self.isplit.filename)
    self.file.seek(self.isplit.offset)
    self.bytes_read = 0
    if self.isplit.offset > 0:
      # read by reader of previous split
      discarded = self.file.readline()
      self.bytes_read += len(discarded)

  def next(self):  # return: (have_a_record, key, value)
    if self.bytes_read > self.isplit.length: # end of input split
      return (False, "", "")
    key = struct.pack(">q", self.isplit.offset+self.bytes_read)
    value = self.file.readline()
    if value == "":  # end of file
      return (False, "", "")
    self.bytes_read += len(value)
    return (True, key, value)

  def getProgress(self):
    return min(float(self.bytes_read)/self.isplit.length, 1.0)
```

L'esempio nel listato 6 mostra come inviare rapporti di stato ed utilizzare i contatori: il metodo `getCounter` del `context` prende come parametri due stringhe di testo: la prima definisce la categoria a cui il counter appartiene, la seconda è il nome del counter; oltre a quelli aggiunti dallo sviluppatore, sono sempre presenti diversi counter di base attivati da Hadoop (figura 3). Il funzionamento di `setStatus` è piuttosto intuitivo: l'unico parametro è una stringa di testo contenente il messaggio che si vuole inviare. Si noti che in questo caso, dovendo effettuare delle inizializzazioni, abbiamo fatto l'override del metodo `__init__` della classe base.

Come esempio di *record reader*, vediamo nel listato 7 una reimplementazione di quello standard di Hadoop, che emette linee di testo come valori ed i corrispondenti offset in byte (rispetto all'inizio del file) come chiavi: attraverso il `context`, il *record reader* ottiene dal framework l'input split, che contiene il percorso del file corrente, l'offset e la lunghezza dello split stesso; utilizzando l'API HDFS, il reader apre il file, si posiziona sull'offset e legge una riga per volta, fermandosi quando il numero di byte letti supera la lunghezza dello split. Il metodo `getProgress` comunica al framework la frazione di input processata.

Nel listato 8, invece, abbiamo un esempio di *record writer*, anche in questo caso una reimplementazione di quello standard, che scrive chiave e valore di output su una riga, separati da tabulazione.

L'esempio mostra come leggere i parametri di configurazione del job (si veda il riquadro 1): in questo caso si tratta di parametri standard di Hadoop, ma lo sviluppatore è libero di definire un numero arbitrario di parametri aggiuntivi. Ad esempio, se volessimo dare all'utente dell'applicazione la possibilità di specificare il carattere di fine riga, potremmo procedere come segue:

```
pu.jc_configure(self, jc, "wordcount.writer.endl", "endl", "\n")
[...]
self.file.write("%s%s%s%s" % (key, self.sep, value, self.endl))
```

Infine, vediamo un esempio di partitioner, ancora una volta una reimplementazione di quello standard:

```
class Partitioner(pp.Partitioner):
  def partition(self, key, numOfReduces):
    return (hash(key) & sys.maxint) % numOfReduces
```

Il framework passa al metodo partition la chiave e il numero totale di reducer; il valore di ritorno è l'indice numerico del reducer a cui inviare la chiave. Utilizzando una funzione di hash, si fa in modo che le chiavi vengano distribuite uniformemente tra i reducer.

Conclusioni

MapReduce è un paradigma di programmazione distribuita che ha i suoi maggiori punti di forza nella scalabilità e nella relativa facilità d'uso. La popolarità di MapReduce è dovuta in gran parte all'esistenza di una robusta implementazione open source, Apache Hadoop, che gode del supporto attivo di Yahoo!

Per ulteriori approfondimenti su Hadoop in generale si può consultare *"Hadoop: The Definitive Guide"* di Tom White (O'Reilly); per quanto riguarda Pydoop, le risorse principali sono il sito web (http://pydoop.sourceforge.net) e *"Pydoop: a Python MapReduce and HDFS API for Hadoop"*, articolo pubblicato nei proceedings di HPDC 2010.

L8 - Record Writer in Pydoop

```
import pydoop.utils as pu

class Writer(pp.RecordWriter):
  def __init__(self, context):
    super(Writer, self).__init__(context)
    jc = context.getJobConf()
    pu.jc_configure_int(self, jc, "mapred.task.partition", "part")
    pu.jc_configure(self, jc, "mapred.work.output.dir", "outdir")
    pu.jc_configure(self, jc, "mapred.textoutputformat.separator",
                                                    "sep", "\t")
    self.outfn = "%s/part-%05d" % (self.outdir, self.part)
    self.file = hdfs.open(self.outfn, "w")

  def emit(self, key, value):
    self.file.write("%s%s%s\n" % (key, self.sep, value))
```

Webografia

- http://wiki.apache.org/hadoop/PoweredBy
  Lista ufficiale di siti ed organizzazioni che utilizzano Hadoop
- http://pydoop.sourceforge.net
  Sito ufficiale di Pydoop
- http://download.oracle.com/javase/6/docs/technotes/guides/jni/index.html
  Documentazione della tecnologia JNI di Oracle
- http://klbostee.github.com/dumbo
  Progetto Dumbo: un wrapper per Streaming che espone un'interfaccia a oggetti
- http://www.crs4.it/distributed-computing
  Gruppo di ricerca per il distributed computing al CRS4
- http://www.boost.org/doc/libs/1_46_1/libs/python
  Librerie Boost-Python
- http://doi.acm.org/10.1145/1851476.1851594
  Documenti sul HPDC 2010