Java Server farm

M. Danelutto

Progetto conclusivo LPRb A.A. 2006-2007

Versione 1.1.1

1 Server farm

Lo scopo del progetto é la realizzazione di un server farm (vedi la definizione di server farm di Wikipedia alla figura 1) dedicato al calcolo, cioé di un server distribuito in grado di calcolare, in base alle richieste dei clienti, una serie di task $\langle x_1, f_1 \rangle, \ldots, \langle x_m, f_m \rangle$ per ottenere una serie di risultati $f_1(x_1), \ldots, f_m(x_m)$ Gli utenti possono connettersi al server farm come ad un normale server e mediante un protocollo opportuno possono comunicare al server un certo numero di coppie funzione/dati $\langle x_i, f_i \rangle$. In conseguenza di tale richiesta riceveranno i risultati $f_1(x_1), \ldots, f_m(x_m)$ secondo le modalitá del protocollo descritto alla sezione 2. Il calcolo dei risultati avviene, nel server farm, utilizzando una serie di macchine opportunamente configurate, ognuna delle quali sará in grado di eseguire uno qualunque dei task proposti dall'utente.
Il progetto si divide logicamente in tre parti:

1. la realizzazione del server vero e proprio, che interfaccia il server farm con l'utente implementando un ben preciso protocollo,
2. la realizzazione del codice relativo al singolo nodo della server farm, cioé di uno dei nodi che, su richiesta del server del punto a) calcola effettivamente i task dell'utente
3. la realizzazione di clienti che permettano il test delle funzionalitá del sistema.

2 Protocollo Server

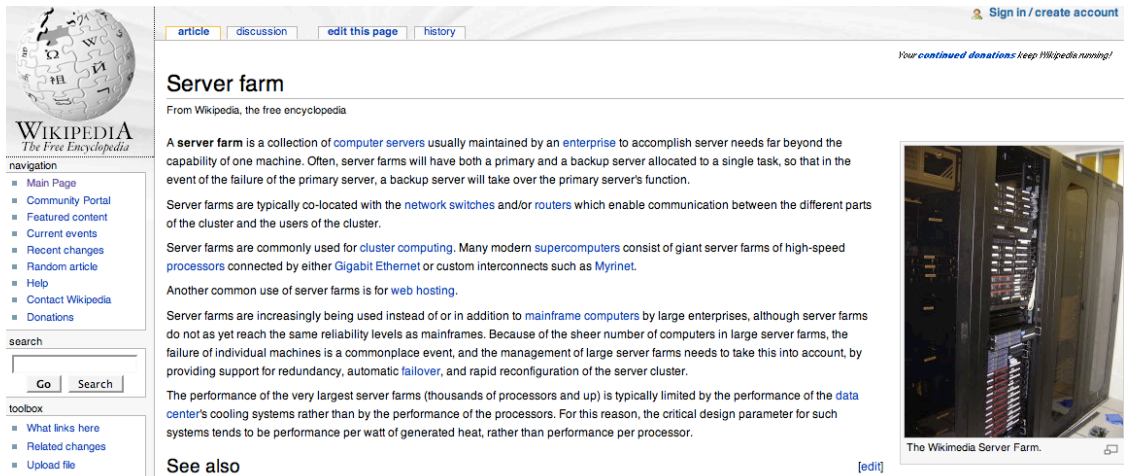
Client e server operano utilizzando un protocollo che prevede l'invio di una richiesta di servizio da parte del cliente e, di conseguenza, l'invio di un pacchetto di risposta da parte del server. Il protocollo é un protocollo ASCII (ovvero tutti i pacchetti di richiesta e risposta sono costituiti da caretteri ASCII) ma é possibile che vengano scambiati anche pacchetti non ASCII per determinate funzionalitá che comunque non coinvolgono i messaggi di richiesta e di risposta. Un client utilizza una connessione con il server per inoltrare una singola richiesta e ricevere la relativa risposta. Un'eventuale richiesta successiva avverrá su una nuova connessione.

Il server riceve messaggi di richiesta sulla porta

`serverfarm.ServerFarm.SERVERFARMPORT`

utilizzando TCP. Il client puó inviare al server una richiesta nel formato:

```
<ReqMsg> ::= SERVERFARM <ReqOp> \n <ReqBody> END \n
<ReqOp>  ::= SERVICE | SERVICEN
<ReqBody>::= <Task> \n | <Task> <ReqBody>
<Task>   ::= FUNCTION <FName> \n DATA <Data>
```



article | discussion | edit this page | history

Sign in / create account

Your continued donations keep Wikipedia running!

WIKIPEDIA
The Free Encyclopedia

navigation
- Main Page
- Community Portal
- Featured content
- Current events
- Recent changes
- Random article
- Help
- Contact Wikipedia
- Donations

search
Go | Search

toolbox
- What links here
- Related changes
- Upload file

Server farm

From Wikipedia, the free encyclopedia

A **server farm** is a collection of computer servers usually maintained by an enterprise to accomplish server needs far beyond the capability of one machine. Often, server farms will have both a primary and a backup server allocated to a single task, so that in the event of the failure of the primary server, a backup server will take over the primary server's function.

Server farms are typically co-located with the network switches and/or routers which enable communication between the different parts of the cluster and the users of the cluster.

Server farms are commonly used for cluster computing. Many modern supercomputers consist of giant server farms of high-speed processors connected by either Gigabit Ethernet or custom interconnects such as Myrinet.

Another common use of server farms is for web hosting.

Server farms are increasingly being used instead of or in addition to mainframe computers by large enterprises, although server farms do not as yet reach the same reliability levels as mainframes. Because of the sheer number of computers in large server farms, the failure of individual machines is a commonplace event, and the management of large server farms needs to take this into account, by providing support for redundancy, automatic failover, and rapid reconfiguration of the server cluster.

The performance of the very largest server farms (thousands of processors and up) is typically limited by the performance of the data center's cooling systems rather than by the performance of the processors. For this reason, the critical design parameter for such systems tends to be performance per watt of generated heat, rather than performance per processor.

The Wikimedia Server Farm.

See also [edit]

Figure 1: Definizione di server farm da Wikipedia

dove `<FName>` rappresenta un nome di funzione[1] e `<Data>` rappresenta uno o piú dati in formato ASCII[2]. Quindi, ad esempio, messaggi di richiesta validi sono:

```
SERVERFARM SERVICE
FUNCTION Inc
DATA 123
END
```

```
SERVERFARM SERVICEN
FUNCTION Inc
DATA 123
FUNCTION Dec
DATA 321
FUNCTION Mul
DATA 123 321
END
```

```
SERVERFARM SERVICE
FUNCTION Mult
DATA 123 321
END
```

Il server risponderá utilizzando pacchetti nel formato:

```
<AnsMsg>  ::= SERVERFARM <AnsType> NUM <Integer>\n <AnsBody> \n END \n
<AsnType>::= OK | NOK
<AnsBody>::= <Ans> \n | <Ans> <AnsBody>
<Ans>     ::= <Summary> RESULT <Data>
            | <Summary> ERROR <Data>
<Summary>::= ANSWER <FName> <Data> HOST <Hostaddress>
```

(dove `<Integer>` rappresenta un intero e `<HostAddress>` rappresenta un indirizzo IP di host) e quindi messaggi di risposta valida (per esempio in corrispondenza dei messaggi di richiesta visti sopra) saranno del tipo:

```
SERVERFARM OK NUM 1
ANSWER Inc 123 HOST fujih23.cli.di.unipi.it RESULT 124
END
```

oppure

```
SERVERFARM NOK NUM 3
ANSWER Inc 123 HOST fujih23.cli.di.unipi.it RESULT 124
ANSWER Dec 321 HOST fujih12.cli.di.unipi.it ERROR class not found
ANSWER Mul 123 321 HOST fujih8.cli.di.unipi.it RESULT 39483
END
```

Riguardo i messaggi di risposta, la presenza di (almeno) un errore fa' si che il messaggio sia etichettato come NOK piuttosto che come OK.

Il client puó anche inviare messaggi "di controllo" secondo la seguente grammatica:

[1] nel nostro caso, il nome di un file `.java` che contiene una classe `FName` che implementa l'interfaccia `Compute`

[2] per esempio, la stringa "123 321" oppure la stringa "12.34" sono `Data` validi

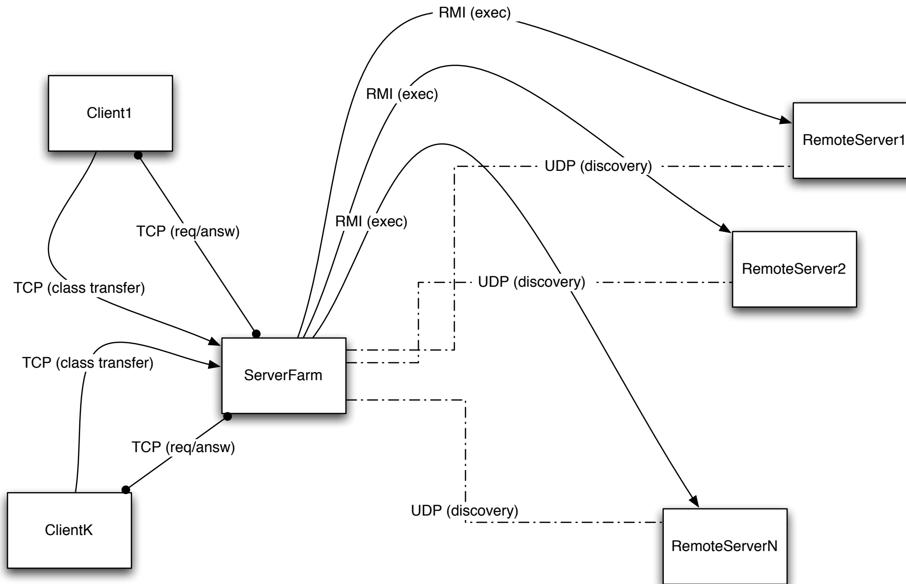


Figure 2: Disegno architetturale del server farm

```
<CtlMsg> ::= SERVERFARM CTL <CtlOp> \n END
<CtlOp>  ::= STATS | SHUTDOWN | RESET
```

Come risposta, il server mander un messaggio composto secondo la grammatica:

```
<CtlAns> ::= SERVERFARM <CtlATy> \n <CtlBody> \n END \n
<CtlATy> ::= REPORT | OK
<CtlBody>::= <Data> \n | <Data> <CtlBody>
```

In particolare, il server inviera´ una risposta di tipo `REPORT` in caso di richiesta di tipo `STATS`. Tale messaggio di risposta conterrá come `<Data>` una serie di linee che contengono i dati relativi al numero, tipo e indirizzo delle macchine "arruolate" nel server farm nonché il numero di task eseguiti fino a quel momento da ciascuno dei server remoti. Ad esempio, un messaggio di tipo `REPORT` potrebbe essere fatto come segue:

```
SERVERFARM REPORT
Host arruolati: 3
Task calcolati: 6034
Tempo medio per task: 2.2333 msec
Utilizzo: 23\%
Host: fujih12.cli.di.unipi.it attivo da Thu Dec  7 08:41:50 CET 2006  2014 task (tempo medio 2.34 msec)
Host: fujih14.cli.di.unipi.it attivo da Thu Dec  7 08:40:33 CET 2006  1997 task (tempo medio 2.14 msec)
Host: fujih12.cli.di.unipi.it attivo da Thu Dec  7 08:40:18 CET 2006  2023 task (tempo medio 2.22 msec)
END
```

La risposta al messaggio di tipo `RESET` o `SHUTDOWN` sará invece un messaggio di `OK` che comprende comunque le statistiche. Nel caso di `RESET` il `ServerFarm` resetterá i contatori delle statistiche (tranne il tempo dell'arruolamento) in modo che ognuno dei contatori riparta, ai fini della richiesta successiva, da zero. Nel caso di `SHUTDOWN` verranno riportate le statistiche relative al momento in cui il `ServerFarm` viene effettivamente terminato.

3 Disegno architetturale del server farm

Lo schema mediante il quale di deve implementare il server farm é quello della figura 2. Un `ServerFarm` accetta le richieste dai clienti e le passa (per l'esecuzione) ad uno dei `RemoteServer`

nel server farm. L'interazione fra cliente e `ServerFarm` avviene mediante socket TCP, quella fra `ServerFarm` e `RemoteServer` avviene mediante UDP (scoperta e arruolamento, raccolta delle statistiche) o RMI (esecuzione remota dei task). In particolare:

- i server interni devono essere realizzati utilizzando RMI. Il processo che interfaccia il server farm con i clienti provvederá a chiamare metodi opportuni dei processi server interni per eseguire i task richiesti
- il processo che interfaccia il server farm con i clienti (`ServerFarm`) deve scoprire utilizzando un meccanismo p2p (basato su UDP e multicast) i server interni (`RemoteServer`) e mantenerne un catalogo. L'indirizzo di multicast da utilizzare sará

  `serverfarm.ServerFarm.MULTICASTGROUP`

  mentre la porta da utilizzare per questo servizio sará identificata dal numero di porta

  `serverfarm.ServerFarm.MULTICASTPORT`
- il processo che interfaccia il server farm con i clienti deve essere realizzato in modo multithreaded (1 thread, possibilmente da un thread pool, per servire ognuna delle richeste cliente)

Tutti i nomi di classi e variabili riportati in Courier in questo documento devono essere usati *esattamente come sono*. Il processo principale del server farm *deve* chiamarsi `ServerFarm` e *deve* far parte del pacchetto di nome `serverfarm`, per esempio, cosí come la classe che implementano i server interni (o remoti) si deve chiamare `RemoteServer`.

4 Codice delle classi che implementano f

Le funzioni f devono essere definite come classi che implementano l'intefaccia `Compute`, cosí definita:

```
public interface Compute {
  public Object exec(Object ... in);
}
```

Ogni funzione nominata nei messaggi di tipo `SERVICE` o `SERVICEN` deve essere presente nella directory in cui viene invocato il cliente. In particolare, per ogni funzione `Fun` nominata in un messaggio di tipo `SERVICE` deve esistere nella directory corrente per il client un file di *Fun.java* o *Fun.class* o *Fun.jar* che contengono il codice che implementa la funzione.

Quando il server riconosce un messaggio di richiesta `SERVICE` con un nome di funzione `Fun`, apre una connessione TCP verso la porta `serverfarm.ServerFarm.CLIENTPORT` del client e si aspetta di ricevere nell'ordine una stringa (caratteri ASCII terminati da un ritorno carrello `\n`) che rappresenta il nome del file e il contenuto del file che implementa la funzione. Il file puó essere un file `.java`, `.class` o `.jar`. dopodiché utilizza tale file per calcolare il task. In caso di richieste di tipo `SERVICEN`, il server si dovrá aspettare di ricevere una sequenza che contiene: il numero (in caratteri ASCII) delle funzioni, seguito da un ritorno carrello `\n`, tante sequenze nome file funzione, spazio, numero di byte del file (in ASCII) ritorno carrello, contenuto del file, quante sono le funzioni.

5 Clienti

Si devono realizzare diversi clienti, in particolare:

- un `Client1` che prende dalla riga di comando il nome della funziona da calcolare e gli eventuali parametri da passare alla funziona ed invia una singola richiesta al server farm per l'esecuzione di quel task
- un `ClientN` che prende dalla riga di comando il nome di un file che contiene un certo numero di righe, ognuna delle quali e' composta dal nome delle funzione e dai parametri, separati da spazi e manda una richiesta `SERVICEN` al server farm con i task trovati nel file

6 Passaggio dei dati per i task

Si assuma che tutti i dati processati nei vari task siano di tipo intero, e che i vari risultati siano ancora di tipo intero. Di fatto questo significa che il tipo delle funzioni dovrá essere:

```
public interface Compute {
  public int exec(int ... in);
}
```

cioé le funzioni sono implementate mediante una chiamata al metodo `exec` che prende uno o piú interi come parametri e restituisce un intero, anziché prendere uno o piú `Object` e restituire un `Object`. Questo semplifica un po' la progettazione e l'implementazione, anche se rende piú banale il tipo di funzioni che si possono utilizzare per il test, ma permette di concentrarsi maggiormente sugli aspetti “di rete” del progetto. Si consideri la possibilitá, per il test del progetto, di realizzare funzioni intere che comunque spendano un po' di tempo per il calcolo, per permettere il test delle caratteristiche di distribuzione e parallelismo del server farm. In particolare, potete pensare di avere funzioni tipo (file `Inc.java`):

```
import serverfarm.Compute;

public class Inc implements Compute {

  /** tempo di "calcolo" della funzione */
  long msecs = 0L;

  /** costruttore
   * @param secs numero di secondi da spendere nel calcolo della funzione<br>
   * attenzione: si puo' usare la sleep solo perche' si assume
   * che un RemoteServer serva una richiesta alla volta ...
   */
  public Inc(int secs) {msecs = secs * 1000L; }

  /** metodo che calcola la funzione
   * @param in valori interi in ingresso. Anche se ne passiamo uno solo
   * dal momento che e' un metodo con numero di parametri int
   * variabile, il parametro si accede come in[0] ...
   * @return il valore calcolato dalla funzione "Int" (nome della claase
   */
  public int exec(int ... in) {
    try {
      Thread.sleep(msecs);
    } catch (InterruptedException e) {} // posso non fare nulla in questo caso
    return(in[0]+1);
  }

}
```

e quindi di creare una funzione `Inc` che impieghi un certo tempo ad essere calcolata, semplicemente passando i secondi da “spendere” nel calcolo al costruttore. Qualora si volesse rendere un po’ pírealistica la cosa, si dovrebbe prevedere che il tempo passato nel costruttore in realtá si solo indicativo del tempo effettivamente passato nel calcolo della funzione, per esempio, variandolo in modo casuale, direttamente all’interno del metodo `exec` di una percentuale, in piú o in meno, del 20%.

7 Test del progetto

Il server farm deve girare su macchine Linux configurate come quelle dell’aula H. Verrá testato su macchine del dipartimento, per ragioni di praticitá, ma in caso di malfunzionamenti, contestazioni o altro fará fede l’esecuzione sulle macchine dell’aula H, effettuata in remoto via terminale con ssh.

8 Relazione e modalitá di consegna

La consegna del progetto avviene nelle date stabilite sulla pagina web del corso e consiste in due parti distinte: la consegna, presso il centralino del dipartimento della relazione del progetto (nella cassetta della posta del docente) e la spedizione per email dei sorgenti e del PDF della relazione (all’email del docente).

La relazione deve contenere:

1. una sezione con la descrizione delle scelte implementative non comprese/descritte in questo progetto, con tre sottosezioni, una per il client, una per il server e una per i server remoti
2. una sezione con *tutti* i comandi necessari per far girare il server farm e il cliente, a partire dal file dei sorgenti come consegnato via email
3. un indirizzo di email per le comunicazioni (eventuali) riguardanti lo stato del progetto

deve essere stampata (non scritta a mano, quindi) e non deve essere piú lunga di 10 fogli A4. La mancanza di una di queste parti, o la sua presentazione in forma incompleta o inutilizzabile o errata comporterá l’impossiblitá di discutere il progetto nella sessione. In particolare, se il progetto non compila o non gira come descritto nella sezione della relazione sulle macchine dell’Aula H, verrá respinto e sará necessaria una nuova consegna nella sessione successiva.

Questo progetto vale per tutte le sessioni dell’anno accademico 2006-2007.

9 Svolgimento del progetto

Il progetto va svolto individualmente. Eventuali realizzazioni di gruppo sono accettate purché dichiarate in fase di consegna e purché la discussione avvenga singolarmente.

10 Facoltativo

Alcune delle parti relative a questo progetto potrebbero essere implementate utilizzando tecnologia standard Java, diversa da quella richiesta per lo svolgimento del progetto. Un esempio per tutti é dato dal caricamento dinamico delle classi. Una volta realizzato il progetto come richiesto dalla specifica fornita nelle sezioni precedenti, lo studente puó prendere in considerazione l’implementazione di varianti del progetto che comportino l’utilizzo di strumenti primitivi della piattaforma Java 1.5 per l’implementazione di parti del progetto che nella soluzione originale sono programmati a piú basso livello, secondo le specifiche date. La consegna di eventuali realizzazioni di questo tipo deve comprendere un’opportuna sezione di documentazione delle scelte implementative e degli strumenti utilizzati.

11 Diff 1.0-1.1

Rispetto alla versione 1.0 sono cambiati i seguenti punti:

- aggiunti ritorno carrello nelle grammatiche dei messaggi. La grammatica completa a questo punto é la seguente:

```
<ReqMsg> ::= SERVERFARM <ReqOp> \n <ReqBody> END \n
<ReqOp>  ::= SERVICE | SERVICEN
<ReqBody>::= <Task> \n | <Task> <ReqBody>
<Task>   ::= FUNCTION <FName> \n DATA <Data>
<AnsMsg> ::= SERVERFARM <AnsType> NUM <Integer>\n <AnsBody> \n END \n
<AsnType>::= OK | NOK
<AnsBody>::= <Ans> \n | <Ans> <AnsBody>
<Ans>    ::= <Summary> RESULT <Data>
           | <Summary> ERROR <Data>
<Summary>::= ANSWER <FName> <Data> HOST <Hostaddress>
<CtlMsg> ::= SERVERFARM CTL <CtlOp> \n END
<CtlOp>  ::= STATS | SHUTDOWN | RESET
<CtlAns> ::= SERVERFARM <CtlATy> \n <CtlBody> \n END \n
<CtlATy> ::= REPORT | OK
<CtlBody>::= <Data> \n | <Data> <CtlBody>
```

- é stato aggiunto, nei messaggi di tipo controllo che il client puó mandare al server farm, il messaggio di tipo `RESET`
- é stato aggiunto un esempio di messaggio di tipo `REPORT`

```
SERVERFARM REPORT
Host arruolati: 3
Task calcolati: 6034
Tempo medio per task: 2.2333 msec
Utilizzo: 23\%
Host: fujih12.cli.di.unipi.it attivo da Thu Dec  7 08:41:50 CET 2006  2014 task (tempo medio 2.34 msec)
Host: fujih14.cli.di.unipi.it attivo da Thu Dec  7 08:40:33 CET 2006  1997 task (tempo medio 2.14 msec)
Host: fujih12.cli.di.unipi.it attivo da Thu Dec  7 08:40:18 CET 2006  2023 task (tempo medio 2.22 msec)
END
```

- si é specificato che i nomi della classi e delle variabili indicati nel progetto devono essere utilizzati alla lettera
- si é specificato che la relazione non deve essere piú di 10 pagine, formato di stampa A4
- si sono aggiunte delle note al paragrafo che spiega come debbono essere effettivamente implementate le "funzioni" compreso un esempio di funzione "pesante".

```
import serverfarm.Compute;
public class Inc implements Compute {
  /** tempo di "calcolo" della funzione */
  long msecs = 0L;
  /** @param secs numero di secondi da spendere nel calcolo della funzione<br>attenzione: si puo'
   * usare la sleep solo perche' si assume che un RemoteServer serva una richiesta alla volta ... */
  public Inc(int secs) {msecs = secs * 1000L; }
  /** metodo che calcola la funzione
   * @param in valori interi in ingresso. Anche se ne passiamo uno solo dal momento che e'
   *  un metodo con numero di parametri int variabile, il parametro si accede come in[0] ...
   * @return il valore calcolato dalla funzione "Int" (nome della claase)    */
  public int exec(int ... in) {
    try {Thread.sleep(msecs);} catch (InterruptedException e) {} // posso non fare nulla in questo caso
    return(in[0]+1);
  }
}
```

12 Diff 1.1-1.1.1

Aggiunto il nome della variabile che contiene il numero di porta da utilizzare per il multicast (`serverfarm.ServerFarm.MULTICASTPORT`). Corretto in tutto il documento il nome del pacchetto (`serverfarm`) che prima appariva nelle due forme `serverFarm` e `serverfarm`.