Comunicazione tra processi con sockets TCP

Su una macchina, un processo può aprire una *server ocket*, ovvero una porta di ascolto, per consentire a processi in esecuzione su altre macchine (o, al limite anche sulla stessa), connesse alla prima attraverso una rete di comunicazione con protocollo IP, di comunicare con il processo stesso. Ad un tale schema si fa spesso riferimento con il termine *client/server*. Infatti, il primo processo solitamente fornisce un servizio, al quale uno o più clienti operanti sulla stessa rete possono accedere. Per poter accedere ad un servizio attivato su una determinata porta, ovvero per poter comunicare con il processo che ha aperto la porta, è necessario specificare l'indirizzo IP della macchina su cui è in esecuzione il processo e la porta sulla quale questo è in ascolto.

Nella figura seguente è illustrata la caso in cui un client invia un messaggio ad un server attraverso il protocollo TCP/IP:

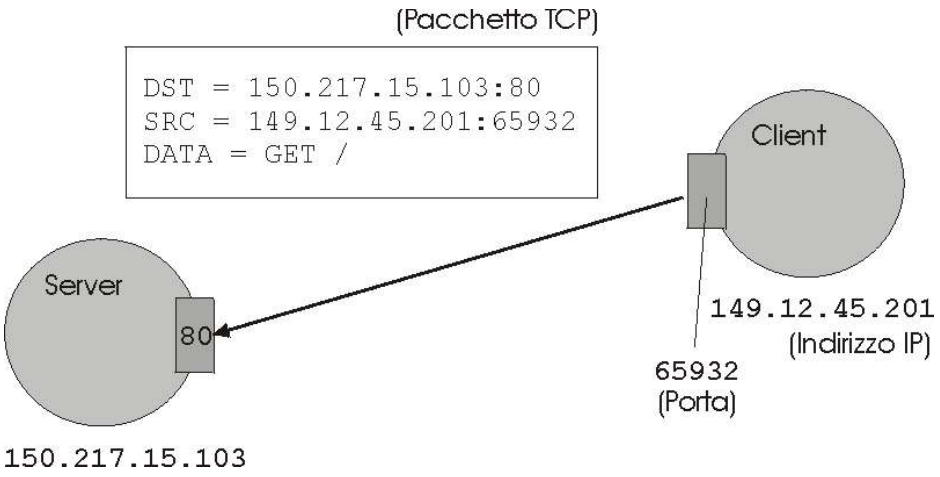

Figura 1: Il client inizia la comunicazione verso il server attivando una porta di comunicazione (*socket*) ed inviando un messaggio alla porta di ascolto sul server (*server socket*)

In particolare, in figura è illustrato l'avvio di una connessione da parte del client (indirizzo e porta della sorgente sono specificati nel pacchetto) verso una porta di ascolto del server. Il processo server si appoggia al sistema operativo per poter aprire una porta in ascolto, specificandone il numero. Tipicamente, per un dato servizio è prevista una porta predefinita. Nel caso dei web server, la porta predefinita è la porta

numero 80, come nell'esempio. Anche il client si appoggia al suo sistema operativo per iniziare una comunicazione con un processo remoto. Il client richiede al sistema una connessione. A questo scopo viene aperta dal sistema una porta, il cui numero, però, non deve essere necessariamente fissato. Infatti, la comunicazione può comunque avvenire perché nel pacchetto inviato dal client al server è specificato, oltre all'indirizzo IP della macchina remota (in maniera tale da consentire il corretto instradamento del pacchetto sulla rete IP) e alla porta su cui gira il processo server, anche la porta aperta sulla macchina client appositamente per consentire questa comunicazione.

Il processo server in ascolto sulla porta specificata accetta la connessione entrante e apre una nuova socket per comunicare direttamente con il client. La server socket, infatti, ha il solo scopo di raccogliere le richieste di connessione entranti. Dalla socket così creata, il server legge la richiesta. Nell'esempio questa è rappresentata dalla stringa "GET /", che, secondo il protocollo HTTP utilizzato per il web, corrisponde alla richiesta della homepage. Il server, una volta interpretata la richiesta, provvede ad inviare la risposta al client. Il messaggio di risposta potrà essere composto da una sequenza di pacchetti TCP (la lunghezza di un pacchetto TCP, infatti, è limitata) indirizzati alla porta specificata dal client al momento della richiesta. Il client potrà quindi leggere i dati e processarli (nel caso di una pagina web, il browser provvederà a visualizzarla).. La risposta è illustrata nella seguente figura:

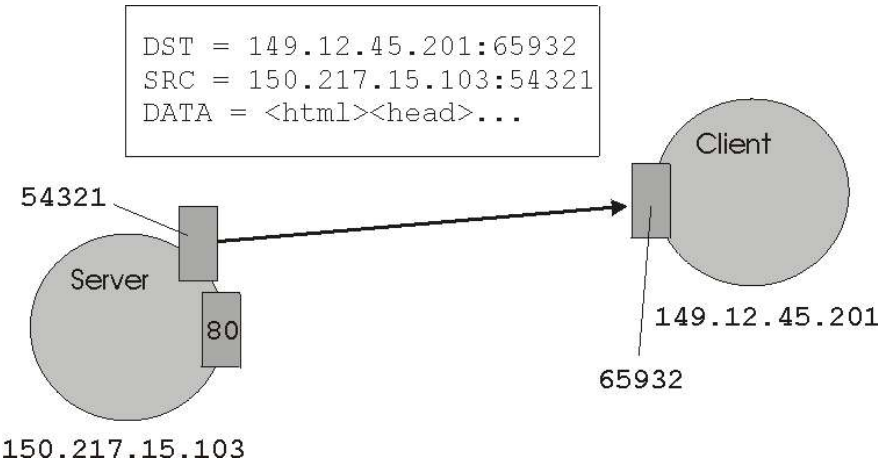

Figura 2: Una volta accettata una richiesta entrante sulla porta di ascolto, il processo server apre una nuova porta per la comunicazione diretta con il client.

Mentre la comunicazione tra server e client procede tra le porte appositamente aperte allo scopo, la porta di ascolto si pone in attesa di altre richieste. Nel caso di un'applicazione ad un singolo thread (o processo), il server non è in grado di soddisfare più di una richiesta per volta. In questo caso è solitamente prevista una coda che raccoglie le richieste entranti.

Al termine della comunicazione, le porte appositamente aperte da server e client devono essere chiuse. Infatti, le porte di comunicazione rappresentano una risorsa limitata per un sistema operativo, e debbono essere a questo restituite quando non sono più necessarie.

Per realizzare una comunicazione tra processi sulla base del protocollo TCP/IP, in Java si possono utilizzare le classi `ServerSocket` e `Socket` (del package `java.net`) per aprire rispettivamente una porta di ascolto e una porta di comunicazione. Due semplici programmi per realizzare un server ed un client sono riportati qui di seguito:

```
package comunicazione.tcp;

/*    classe Java che realizza un semplice server TCP.
      Il server legge i caratteri inviati dal client, e li rimanda indietro.
*/

import java.io.InputStream;
```

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.ServerSocket;

public class TCPEchoServer {
      public static void main( String args[] ) {
            ServerSocket server_socket = null;
            int porta = Integer.parseInt( args[ 0 ] );
            try {
            /*    crea la porta su cui ascoltare istanziando un oggetto
                  della classe ServerSocket. Al costruttore viene passato
                  l'identificatore della porta su cui si intende porre in
                  ascolto il processo per fornire il servizio
            */
                  server_socket = new ServerSocket( porta );
            } catch ( IOException ioe ) {
                  System.err.println( "impossibile creare socket" );
                  System.exit( 1 );
            }
            System.out.println( "server attivo sulla porta " + porta );
            Socket client_socket = null;
            boolean esegui = true;
            while ( esegui ) {
                  /*    il server si blocca in ascolto sulla porta finche' non
                        arriva una richiesta da parte in un client. Il metodo
                        restituisce quindi un oggetto di tipo socket, attraverso
                        il quale il server puo' comunicare con il client
                  */
                  try {
                        client_socket = server_socket.accept();
                        /*    ottiene lo stream di ingresso collegato alla
                              socket, ovvero il canale attraverso il quale il
                              client invia la richiesta al server
                        */
                        InputStream is = client_socket.getInputStream();
                        /*    ottiene lo stream di uscita collegato alla socket,
                              ovvero il canale attraverso il quale il server
                              invia la risposta al client
                        */
                        OutputStream os = client_socket.getOutputStream();
                        /*    legge i dati sullo stream di ingresso e li ricopia
                              sullo stream di uscita
                        */
                        boolean stop = false;
                        while ( ! stop )
                        {
                              int b = is.read();
                              if ( -1 == b )
                                    stop = true;
                              else
                                    os.write( (byte) b );
                        }
                  } catch ( IOException ioe ) {
                        System.err.println( "errore di I/O" );
                  } finally {
                        try {
                              client_socket.close();
                        } catch ( IOException ioe ) {}
                  }
            }
            /*    si chiude la socket (anche se in pratica questa istruzione non
                  verra' eseguita perche' non e' prevista la possibilita' di
```

```
                uscire dal ciclo infinito di servizio)
            */
            try {
                server_socket.close();
            } catch ( IOException ioe ) {}
        }
}
```

Per lanciare il server:

```
java -cp . comunicazione.tcp.TCPEchoServer <porta>
```

e per verificarne il funzionamento:

```
telnet localhost <porta>
```

e quindi digitare caratteri. Nota bene: questo server non è in grado di servire più richieste contemporaneamente.

Il seguente programma accede al servizio di echo. In particolare, esso crea una socket per connetteri al servizio e vi invia una stringa (fornita come argomento a linea di comando). La risposta del server viene visualizzata sullo schermo.

```
package comunicazione.tcp;

/**
*           classe Java che realizza un semplice client TCP.
*/

import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;

public class TCPEchoClient {
      public static void main( String args[] ) {
            Socket socket = null;
            try {
                  String host = args[ 0 ];
                  int porta = Integer.parseInt( args[ 1 ] );
                  socket = new Socket( host, porta );
                  InputStream is = socket.getInputStream();
                  OutputStream os = socket.getOutputStream();
                  /*    copia in un vettore di bytes la rappresentazione
                        della stringa che si intende inviare al server
                  */
                  byte[] dati = args[ 2 ].getBytes();
                  /*    invia i dati al server
                  */
                  os.write( dati );
                  /*    chiude lo stream per l'invio dei dati al server
                        (la richiesta e' completata)
                  */
                  socket.shutdownOutput();
```

```
                boolean stop = false;
                while ( ! stop )
                {
                      int b = is.read();
                      if ( -1 == b )
                      {
                            /*    e' stata raggiunta la fine dello stream
                            */
                            stop = true;
                      }
                      else
                      {
                            System.out.print( (char) b );
                            System.out.flush();
                      }
                }
                socket.close();
          } catch ( UnknownHostException uhe ) {
                System.err.println( "host sconosciuto" );
          } catch ( IOException ioe ) {
                System.err.println( "errore di I/O" );
          } finally {
                try {
                      if ( null != socket )
                            socket.close();
                } catch ( IOException ioe ) {}
          }
     }
}
```

Per lanciare il client:

```
java -cp . comunicazione.tcp.TCPEchoServer <host> <porta> <stringa>
```

Multiprogrammazione e sincronizzazione nella realizzazione di servizi TCP

Un semplice servizio TCP potrebbe essere realizzato secondo lo schema riportato qui di seguito (in blu sono evidenziate le primitive Java che rendono possibile la comunicazione tra client e server tramite sockets TCP)[1]:

```
//   creazione della porta predefinita di ascolto
ServerSocket server = new ServerSocket( port );
//   indefinitamente, accettazione e conseguente gestione richieste entranti
while ( true ) {
      //  accettazione richiesta entrante, proveniente da un cliente
      Socket client = server.accept();
      //  creazione degli streams di I/O per comunicazione con cliente
      InputStream is = client.getInputStream();
      OutputStream os = client.getOutputStream();
      /*  lettura dati dallo stream di ingresso e scrittura sullo stream di
          uscita per la realizzazione di un servizio di echo, fino a chiusura
          della connessione da parte del cliente  */
      int data = is.read();
      while ( -1 != data ) {
            os.write( (byte) data );
            data = is.read();
      }
      //  chiusura degli streams di I/O
      is.close();
      os.close();
      //  chiusura della connessione
      client.close();
}
```

Le principali caratteristiche di questa soluzione sono:

- un singolo thread per la gestione delle richieste provenienti dai clients
- utilizzo di primitive bloccanti per le operazioni di I/O (evidenziate in rosso nel codice riportato sotto)

```
while ( true ) {
    Socket client = server.accept();
    InputStream is = client.getInputStream();
    OutputStream os = client.getOutputStream();
    int data = is.read();
    while ( -1 != data ) {
           os.write( (byte) data );
           data = is.read();
    }
    is.close();
    os.close();
    client.close();
}
```

1 Si osservi che nell'implementazione riportata, la lettura e la scrittura dei dati avviene un carattere alla volta; questa soluzione non garantisce in generale prestazioni soddisfacenti, per cui si consiglia di utilizzare metodi per la lettura e la scrittura che operino il trasferimento di blocchi di dati.

Queste caratteristiche fanno sì che la gestione delle singole richieste eventualmente inoltrate da più clienti sia effettuata secondo uno schema FIFO (ogni iterazione sul ciclo più esterno rappresenta la gestione di un singolo cliente)

Come conseguenza dello gestione FIFO si può verificare “l'effetto convoglio”. Ad esempio:

- Richiesta di trasferimento di 1MB di dati
- 3 clienti accedono attraverso modem 56kb/s
  tempo di trasferimento = 146,2" circa
- 1 cliente accede attraverso LAN 100Mb/s
  tempo di trasferimento = 0,08"

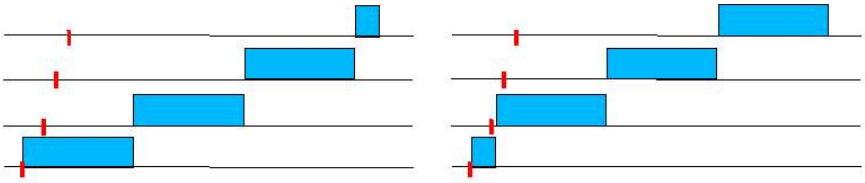

Figura 3: esemplificazione dell'effetto convoglio nel caso di servizio realizzato con singolo thread e primitive di I/O bloccanti; a sinistra, il cliente con connessione veloce, arrivando per ultimo, deve attendere il completamento della gestione delle richieste precedenti, effettuate da clienti dotati di connessioni lente.

Come si evince dalla figura riportata sopra, a parità di throughput, il tempo di attesa (e il tempo di risposta) per il cliente “veloce” cambia notevolmente a seconda dell'ordine di arrivo.

La soluzione presentata, inoltre, risulta non applicabile nei casi in cui i clienti aprono delle connessioni illimitate nel tempo (o comunque con durata superiore ai tempi massimi di attesa tollerati dagli altri clienti).

La soluzione appare pertanto troppo limitata. Infatti, i servizi TCP devono tipicamente gestire le richieste provenienti da diversi clienti, per cui è evidente che la realizzazione di tali servizi potrebbe avvantaggiarsi dell'adozione delle tecniche della multiprogrammazione (e quindi anche della sincronizzazione e della comunicazione

tra processi e/o threads). Nel seguito saranno discussi alcuni modelli per lo sviluppo di tali servizi, che combinano in modo diverso le tecniche di multiprogrammazione.

Per conferire alla discussione una maggiore concretezza, questi modelli saranno discussi, oltreché da un punto di vista generale, anche nel caso particolare del web server Apache. Il programma Apache è infatti uno dei più diffusi programmi per la pubblicazione di siti web, ovvero per l'erogazione di documenti ipertestuali tramite protocollo HTTP. Nell'ambito del corso esso risulta essere particolarmente interessante perché, disponendo di diversi moduli per la multiprogrammazione (MultiProgramming Modules, o MPM), consente di analizzare come possano essere combinate nella pratica le diverse tecniche della multiprogrammazione per adeguarsi ai diversi contesti operativi e a specifici requisiti in termini di prestazioni, affidabilità e sicurezza.

Storicamente i diversi moduli di programmazione sono stati introdotti per consentire lo sviluppo di Apache su sistemi operativi diverso da quello originale (Unix), che offrono strumenti diversi per la multiprogrammazione. Successivamente, con lo sviluppo della versione 2.0, si è riconosciuto che i diversi moduli, oltreché per consentire la portabilità dell'applicazione su piattaforme diverse, potevano essere utilizzati per soddisfare, su una medesima piattaforma, esigenze diverse in termini di prestazioni, affidabilità e sicurezza.

In generale, lo sviluppo di un servizio TCP richiede la definizione di un'architettura per la multi-utenza, sovente ottenuta mediante la multiprogrammazione, e quindi comporta determinare come creare e controllare le unità operative (tasks) e come comunicare le richieste da elaborare. Le principali variabili di progetto sono:

- modello di elaborazione
- processi e/o threads
- dimensione dell'insieme di unità operative (fissa o dinamica)

- modalità di comunicazione tra le unità operative
- sospensione le unità operative inattive (variabili di condizione, semafori, ...)
- ascolto delle connessioni entranti (numero di unità in ascolto)
- ...

Vantaggi e svantaggi connessi ad alcune possibili scelta relativamente alle variabili di progetto sopra elencate saranno discussi in dettaglio nel seguito.

Gatekeeper[2]

Lo schema più semplice ed intuitivo per introdurre la multiprogrammazione nella realizzazione di un servizio TCP prevede che il thread (o processo) principale del server si ponga in ascolto sulla porta predefinita e ad ogni nuova richiesta proveniente da un cliente provvede a creare, in un flusso di esecuzione separato (thread o processo) un gestore per la richiesta (*request handler* o, più semplicemente, *handler*) cui sarà delegata la gestione della specifica richiesta (lettura della richiesta e generazione della risposta). Il principio di funzionamento è sintetizzato in figura 4; in figura 5 è invece rappresentato il comportamento dinamico del servizio via via che si presentano le richieste dei diversi clienti.

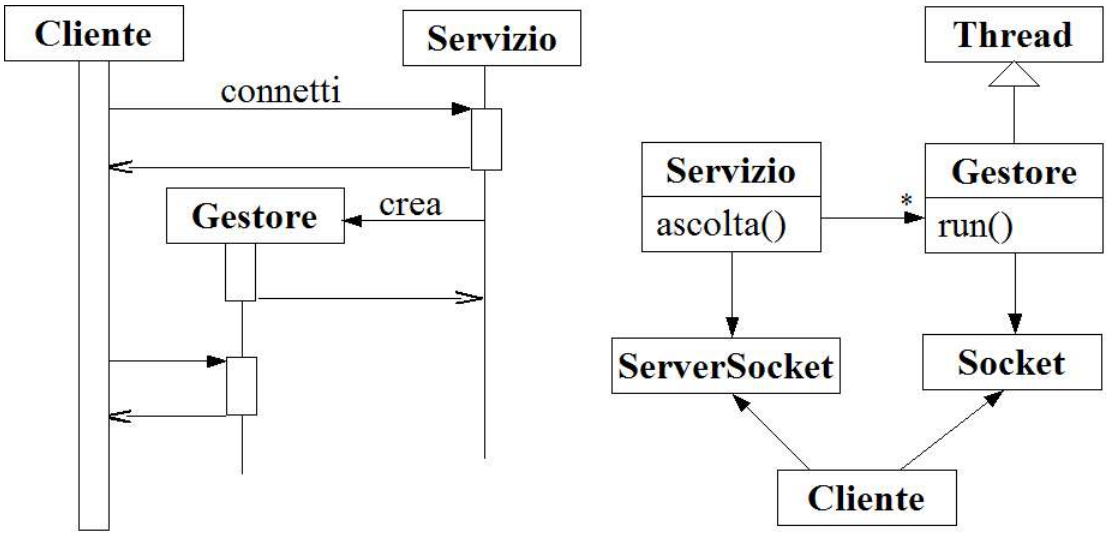

Figura 4: a sinistra, il diagramma di sequenza che illustra la successione di azioni che portano alla gestione di una richiesta da parte di un oggetto *Handler*; a destra, il diagramma delle classi utilizzate per la soluzione di tipo gatekeeper.

2 v. anche design pattern Acceptor-Connector

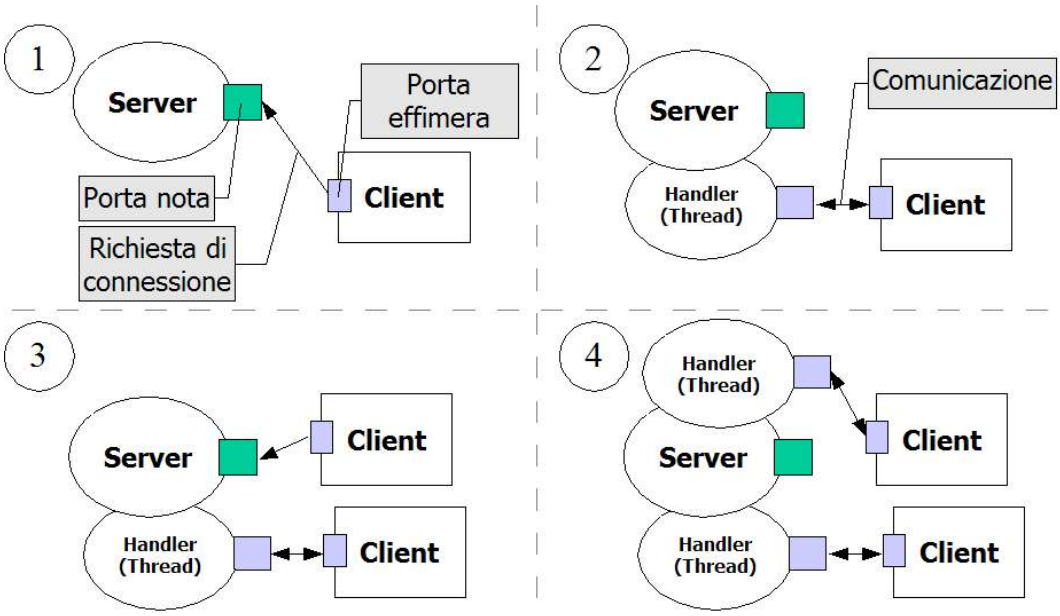

Figura 5: Comportamento dinamico di un servizio TCP realizzato tramite multiprogrammazione, ed in particolare secondo uno schema di tipo gatekeeper.

Nel caso di un programma Java che fa uso di threads, il codice per l'accettazione delle richieste potrebbe essere il seguente:

```
ServerSocket server_socket = new ServerSocket( port );
while( true ) {
      Socket client_socket = server_socket.accept();
      RequestHandler handler = new RequestHandler( client_socket );
      handler.start();
}
```

mentre gestore potrebbe essere sviluppato secondo il seguente schema:

```
public class RequestHandler extends Thread {
      public RequestHandler ( Socket s ) {
            clientSocket = s;
      }

      public void run() {
            InputStream is = clientSocket.getInputStream();
            OutputStream os = clientSocket.getOutputStream();
            int data = is.read();
            while ( -1 != data ) {
                  os.write( (byte) data );
                  data = is.read();
            }
            is.close();
            os.close();
            clientSocket.close();
      }

      private Socket clientSocket = null;
}
```

La soluzione appena esaminata risulta essere molto semplice per i seguenti motivi:

- non richiede l'adozione di particolari meccanismi di sincronizzazione
- la comunicazione tra thread principale e gestore avviene solo in fase di inizializzazione del gestore (per comunicare i dati relativi al cliente); una volta inizializzato il gestore non è quindi prevista alcuna forma di comunicazione né tra flusso principale e gestore, né fra i diversi gestori (le richieste sono indipendenti).

Nel caso del web server Apache, limitatamente ai sistemi operativi Unix/Linux, questa modalità è realizzara attraverso il programma di sistema `inetd`, che opera come "*gatekeeper*". Il programma `inetd`, infatti

- è registrato per un servizio, ovvero è in ascolto sulla porta associata al servizio;
- a fronte di una richiesta genera un nuovo processo tramite la primitiva `fork()`;
- se il programma che deve gestire la richiesta è diverso dal gatekeeper (come nel caso di Apache), si esegue anche una `exec()`.

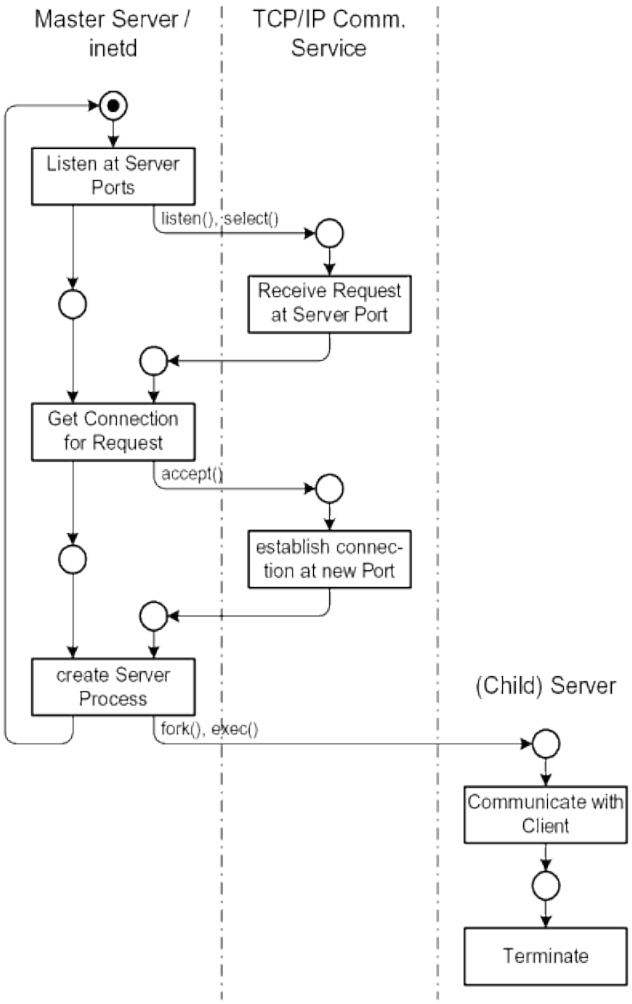

Figura 6: diagramma che illustra la successione temporale delle operazioni nel caso del web server Apache operato tramite gatekeeper.

Lo schema appena presentato richiede la creazione di un nuovo thread/processo per ogni nuova richiesta, e la terminazione dello stesso non appena è completata la gestione della richiesta. Considerati i tempi necessari allo svolgimento di queste operazioni (in particolare nel caso dei processi), lo schema risulta vantaggioso solo se la gestione si estende su un arco di tempo sufficientemente ampio o se tra successivi cicli richiesta/risposta deve essere mantenuta un'informazione di stato.

La gestione di un servizio HTTP con Apache tramite `inted` è pertanto sconsigliata sia perché HTTP è un protocollo senza stato, con cicli di richiesta/risposta molto brevi, sia perché il programma Apache, essendo particolarmente complesso e richiedendo numerose risorse, comporta tempi lunghi

per la creazione e l'avvio di un nuovo processo, e quindi le prestazioni del sistema ne risentono negativamente.

Inoltre, qualora tra la ricezione della richiesta da parte del gatekeeper e la creazione di un nuovo thread/processo sia richiesta l'esecuzione di un numero elevato, si può avere un intervallo di tempo abbastanza lungo durante il quale il gatekeeper non può tornare sulla porta di ascolto.

Infine, nel caso non si preveda un limite superiore al numero di richieste che possono essere gestite concorrentemente, esiste il rischio che nel caso di un elevato numero di richieste i threads/processi dedicati alla gestione delle medesime esauriscano le risorse del sistema, con un conseguente degrado delle prestazioni (ad esempio, eccessivo swapping a causa dell'esaurimento della memoria fisica).

Task pooling

Per ovviare ai suddetti problemi legati alla creazione/terminazione di threads/processi, particolarmente nei contesti applicativi per i quali si ha un elevato tasso di richieste la cui gestione richiede un tempo molto limitato, è stato introdotto il concetto di *task pool*: questo è costituito da un insieme limitato di unità operative (threads o processi) precostituite cui viene delegata la gestione delle risorse.

All'avvio del servizio il thread/processo principale (*master*) provvede a creare altri threads/processi che costituiranno l'insieme delle unità operative, e alle quali in seguito verrà delegata la gestione delle singole richieste. A differenza dello schema precedente, terminata la gestione di una richiesta, un'unità operativa si rende nuovamente disponibile per la gestione di una nuova richiesta (si realizza, cioè, un riuso delle unità operative).

A regime il thread/processo principale svolgerà compiti di supervisione (es. verificare lo stato delle unità operative, misurare il carico del servizio) e controllo (es. aumentare/diminuire il numero di unità operative in funzione del carico del servizio, per garantire delle buone prestazioni e, al contempo, un utilizzo ottimale

delle risorse). L'utilizzo del task pool fa sì che in fase di esecuzione non si incorre nei costi di generazione e terminazione delle unità operative.

In definitiva, lo schema di principio per il *master* potrebbe essere il seguente:

```
creazione N unità operative (threads/processi)
supervisione e controllo del servizio
```

Mentre quello di un unità operativa potrebbe essere:

```
while(true) {
      lettura nuova richiesta
      elaborazione della richiesta
}
```

Questo schema di base deve essere raffinato per consentire l'accettazione delle richieste ed il relativo smistamento alle diverse unità operative. In particolare, si possono individuare almeno 2 schemi alternativi:

- *leader-follower*, in cui ogni unità operativa preposta all'elaborazione delle richieste provvede autonomamente a leggere la richiesta che poi provvederà ad elaborare;
- *job-queue* (o produttore consumatore), in cui le richieste vengono raccolte da unità operative dedicate (*listeners*), che poi le distribuiscono alle unità operative preposte all'elaborazione (*workers*).

Leader-follower

Qualora ogni unità operativa preposta all'elaborazione abbia anche la responsabilità leggere autonomamente la richiesta proveniente dal cliente, l'unità operativa deve poter accedere alla socket relativa alla porta di ascolto predefinita. Poiché quest'ultima è una risorsa cui si deve accedere in mutua esclusione, lo schema di funzionamento dell'unità operativa potrebbe essere così raffinato:

```
while(true) {
      accesso alla sezione critica per la server socket
            lettura nuova richiesta dalla porta di ascolto
      rilascio della sezione critica per la server socket
      elaborazione della richiesta
}
```

Da un punto di vista dell'elaborazione, un'unità operativa può essere impegnata (*busy*) nella gestione di una richiesta, oppure inattiva (*idle*); le unità inattive sono in competizione per l'accesso alla porta di ascolto, e fra queste al più una (detta *leader*) si può trovare in ascolto sulla porta (ovvero all'interno della sezione critica), mentre le altre si trovano accodate in attesa di poter accedere alla sezione critica (e per questo sono dette *followers*).

Più concretamente, un semplice servizio echo operante secondo lo schema *leader-follower* ed implementato in Java potrebbe essere sviluppato come segue:

```
public class Task extends Thread {
      public class Task ( ServerSocket s ) {
            serverSocket = s;
      }
      public void run()
      {
            while(true)
            {
                  try
                  {
                        Socket client = null;
                        synchronized( server )
                        {
                              client = server.accept();
                        }
                        if( null != client )
                        {
                              elabora( client );
                              client.close();
                        }
                  }
                  catch( IOException ioe  )
                  {
                        System.out.println( ioe.toString() );
                  }
            }
      }
      ...
      private ServerSocket serverSocket = null;
}
```

Preforking

Il modello *preforking* del web server Apache segue lo schema leader-follower:

- all'avvio il master genera i processi figli
- ogni figlio ha un solo thread
- a regime, il master controlla il numero di processi intattivi per determinare dinamicamente la dimensione del pool

- i figli si mettono in ascolto e gestiscono le richieste
- l'accesso alla porta di ascolto avviene in mutua esclusione, secondo lo schema (o pattern) leader/follower

L'intero meccanismo è sintetizzato nella figura seguente.

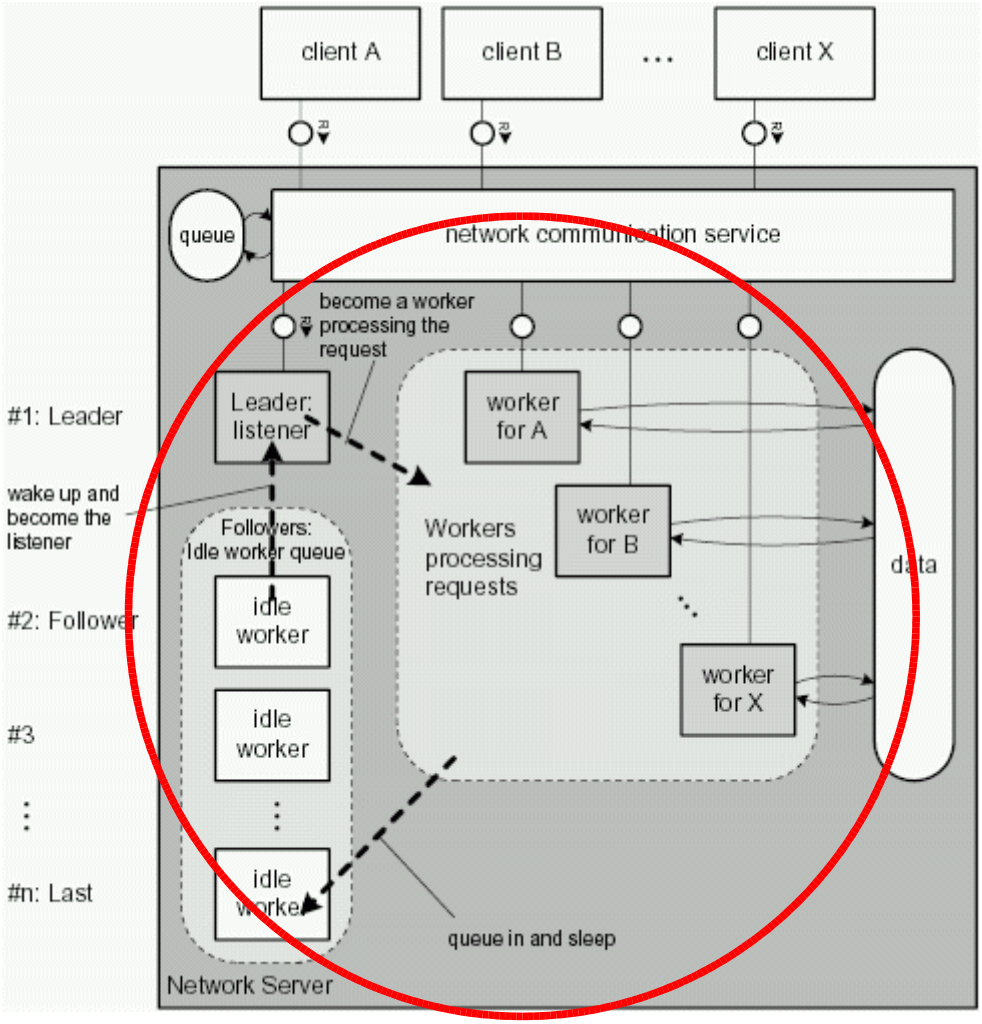

Figura 7: diagramma che sintetizza l'interazione tra i diversi componenti del web server Apache nel caso in cui operi secondo lo schema *preforking*.

Job queue

Lo schema alternativo al *leader-follower* prevede una netta separazione tra la ricezione e l'elaborazione delle richieste attraverso la definizione di due distinte classi di unità operative: le unità della prima classe (*listeners*) provvedono a prendere in carico dalla porta di ascolto le richieste provenienti dai clienti, mentre quelle della seconda classe (*workers*) si occupano dell'elaborazione delle richieste. L

scambio delle informazioni tra le due classi avviene tramite una coda di lavori (*job queue*) in cui un'unità di tipo *listener* deposita la richiesta, e da cui un'unità inattiva di tipo *worker* la preleva per elaborarla. In pratica si tratta di uno schema del tipo produttore-consumatore.

Considerato che l'accesso alla porta di ascolto richiede la mutua esclusione, e che in genere l'accettazione di una richiesta richiede l'esecuzione di operazioni di durata complessiva molto limitata, l'implementazione tipica prevede un solo produttore (unità di tipo *listener*) e molti consumatori (unità di tipo *worker*).

Non si esclude comunque la possibilità di avere più produttori (ad esempio se il servizio deve essere accessibile attraverso più porte, o se i tempi legati all'accettazione delle richieste non sono trascurabili o comunque non sono compatibili con stringenti requisiti in termini di prestazioni); in tal caso, è richiesta la sincronizzazione nell'accesso alla porta di ascolto da parte dei diversi produttori.

Lo schema richiede l'adozione di adeguati meccanismi di sincronizzazione per l'accesso alla coda—come del resto è ovvio per un problema di tipo produttore-consumatore.

```
public class Servizio extends Thread {

      private int porta;
      private JobQueue socketbuf = null;

      public Servizio ( int p, JobQueue jq ) {
            porta = p;
            socketbuf = jq;
      }

      public void run() {
            ServerSocket server = new ServerSocket( portnum );
            while( true ) {
                  Socket client = null;
                  try {
                        server.accept();
                        socketbuf.inserisci( client );
                  } catch( Exception e ) {
                        System.out.println( e.toString() );
                        if ( null != client )
                              client.close();
                  }
            }
      }

public class Task extends Thread {
      public Task ( BufferCircolareJob sb )
```

```
        {
                socketbuf = sb;
        }

        public void run()
        {
                while( true )
                {
                        try
                        {
                                Job job = socketbuf.preleva();
                                elabora( job );
                        }
                        catch( Exception e )
                        {
                                System.out.println( e.toString() );
                        }
                }
        }
        ...
        private BufferCircolareJob socketbuf;
}
```

La coda dei lavori è utilizzata da diversi moduli di Apache, che sono descritti brevemente nel seguito.

WinNT (Apache 1.3)

- multiprogrammazione realizzata con i threads (anziché processi, come invece era prassi comune nella versione 1.3 di Apache per Unix)
- numero fisso di figli, considerato che i threads inattivi non influenzano sensibilmente le prestazioni
- 2 processi:
  - supervisor
  - worker (che comprende 3 tipi di threads: master, worker, listener)
- job queue

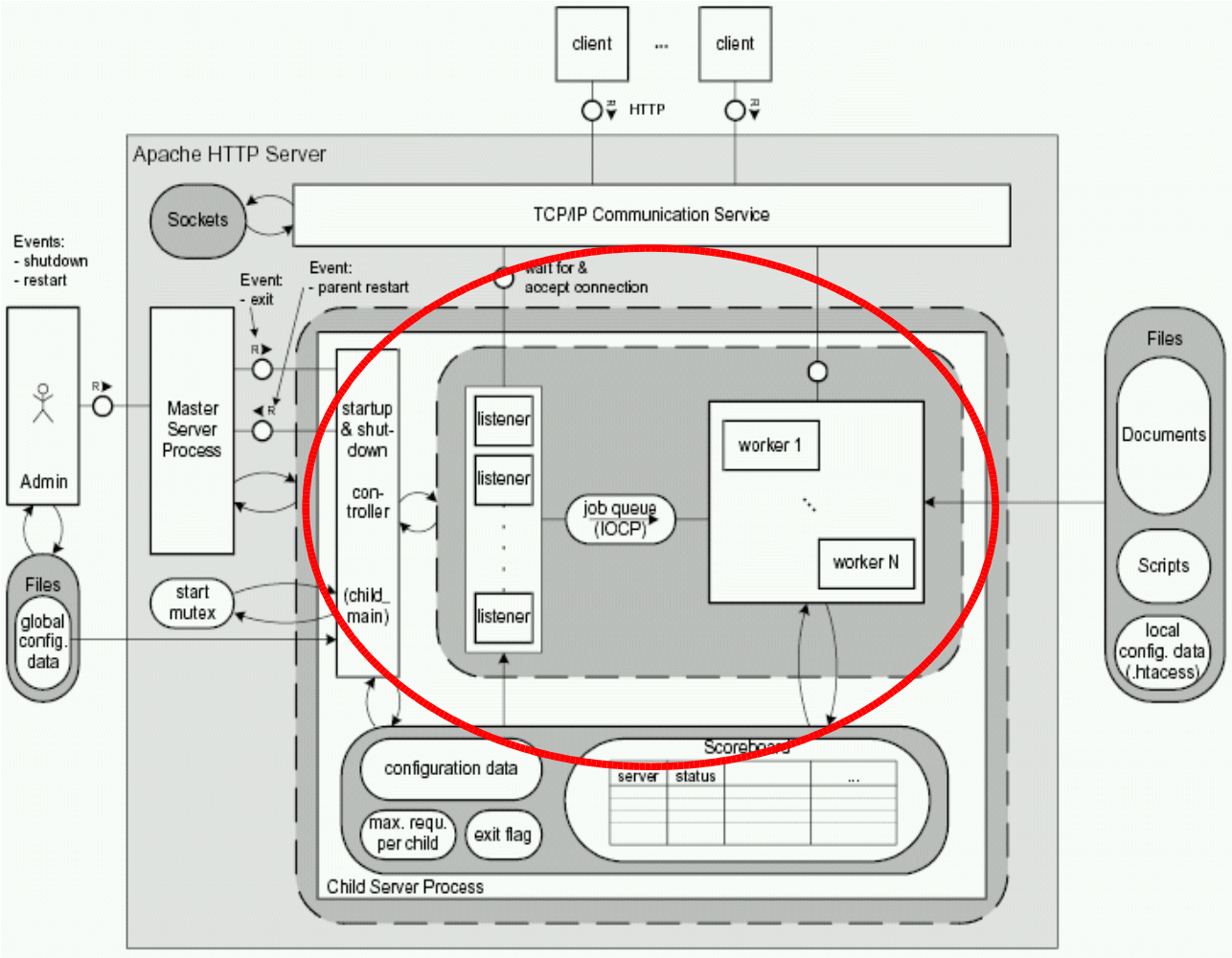

Figura 8: diagramma che sintetizza l'interazione tra i diversi componenti del web server Apache nel caso in cui operi tramite *job queue (WinNT)*.

Worker (Unix/Linux)

- mix di processi e threads
- numero di processi variabile
- numero di threads per processo fisso
- job queue a livello di thread
- thread listener concorre per la porta di ascolto solo se ci sono threads in *idle queue*
- combina stabilità di MP e prestazioni di MT

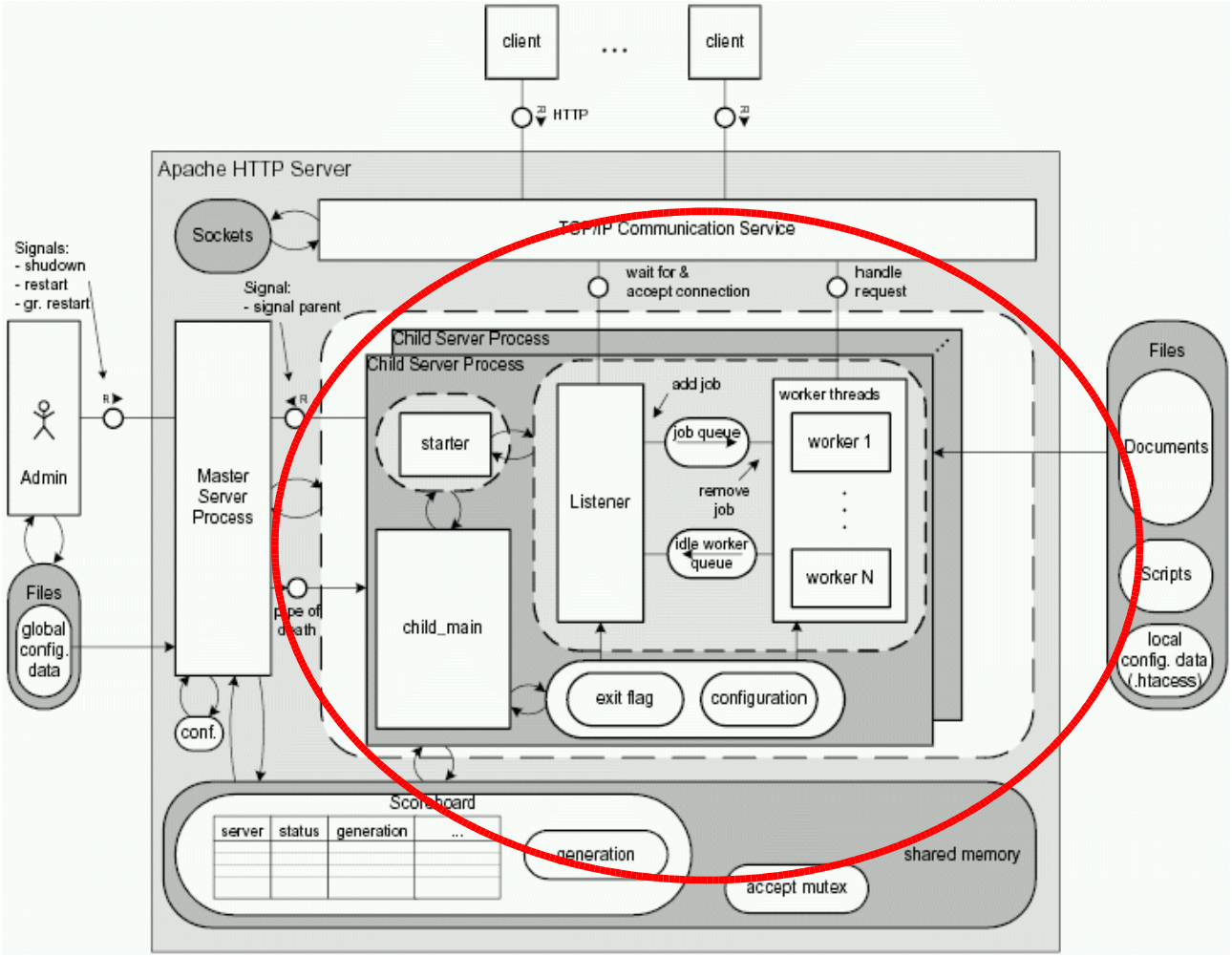

Figura 9: diagramma che sintetizza l'interazione tra i diversi componenti del web server Apache nel caso in cui operi tramite *job queue (Worker)*.

Leader (sperimentale)

E' una variante di *worker* che adotta il pattern leader/follower sia a livello di processo che a livello di thread:

- nessun ritardo per l'inoltro della richiesta
- soltanto un thread viene risvegliato (non c'è competizione per la mutua esclusione)
- completata la risposta, i thread inattivi vengono posti su una pila LIFO (riducendo così lo swapping)

Per child (sperimentale)

E' un'altra variante di worker, che però prevede

- numero fisso di processi
- numero variabile di threads
- si può impostare un UID diverso per ogni processo

Riferimenti bibliografici

http://httpd.apache.org/docs-2.0/

http://apache.hpi.uni-potsdam.de/document