

Architetture dei server TCP

materiale didattico integrativo per il corso di
Architetture software per applicazioni in rete
Master in Multimedia – Internet Engineering
a.a. 2003/2004

Ing. Jürgen Assfalg

TCP/IP in Java

Java API per programmazione sockets

- `java.net`
 - `Socket`
 - `ServerSocket`
- `java.io`
 - `InputStream`
 - `OutputStream`

Semplice server TCP in Java

```
ServerSocket server = new ServerSocket( port );
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();
}
```

SEchoServer

Semplice server TCP in Java

Caratteristiche della soluzione

- I/O bloccante
- singolo thread
- gestione delle richieste secondo uno schema FCFS (First Come First Served)

Semplice server TCP in Java

```
ServerSocket server = new ServerSocket( port );  
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();  
}
```

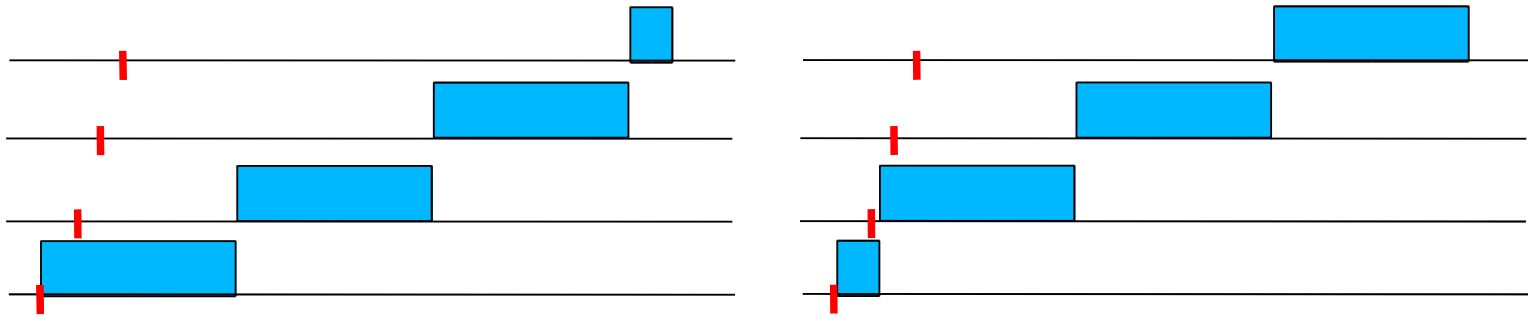
Problemi

Si può verificare l' "effetto convoglio"

Esempio:

- Richiesta di un 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"

Problemi



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

Alternative

Possibili soluzioni

- server multi-threaded
- I/O non bloccante
- I/O multiplexing (o readiness selection)

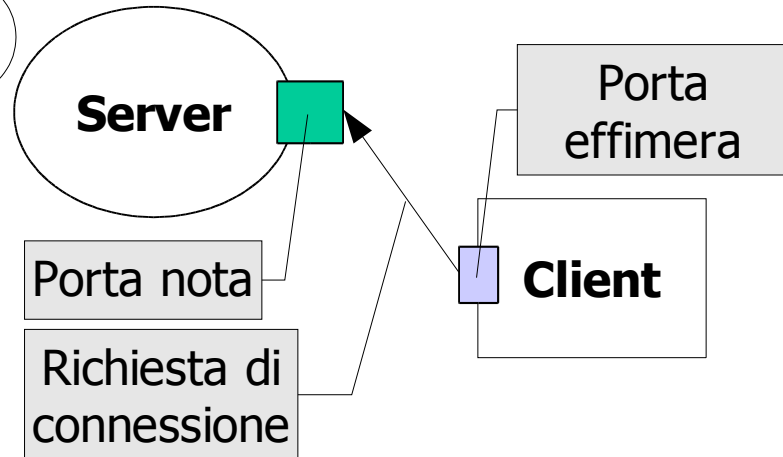
Server TCP multi-threaded

Principio di funzionamento

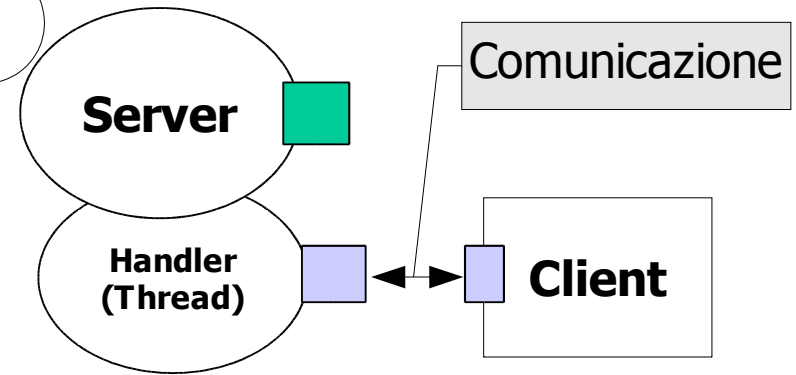
- il thread principale accetta le connessioni provenienti dai client; per ogni nuova connessione genera un nuovo thread
- ciascun thread si occupa della lettura di una singola richiesta e della generazione della relativa risposta

Server TCP multi-threaded

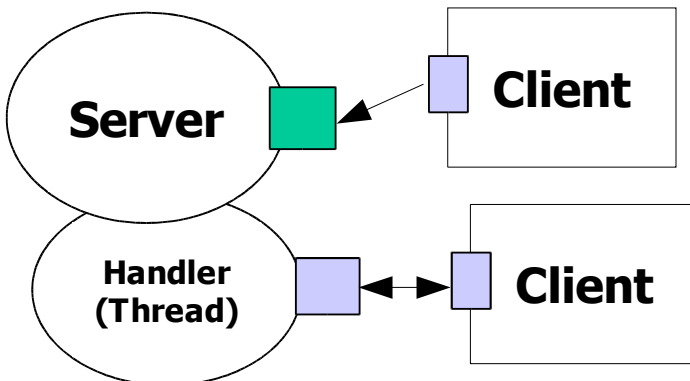
1



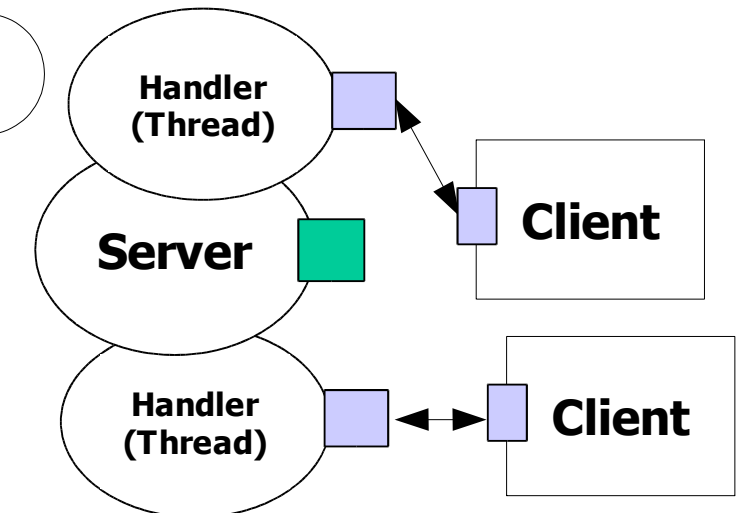
2



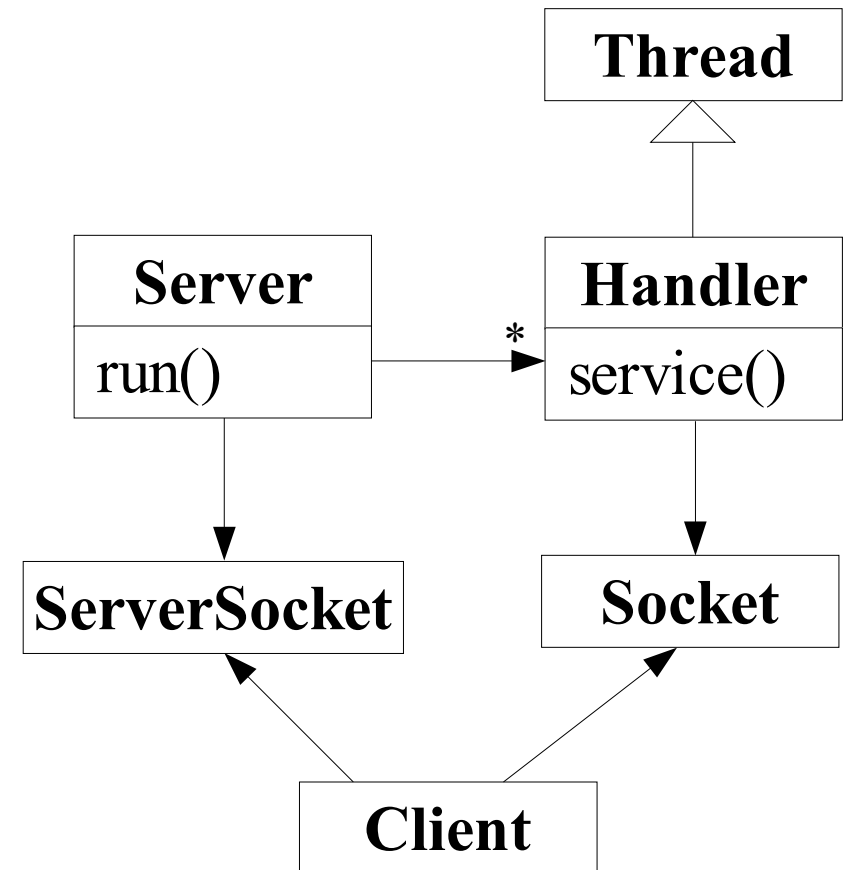
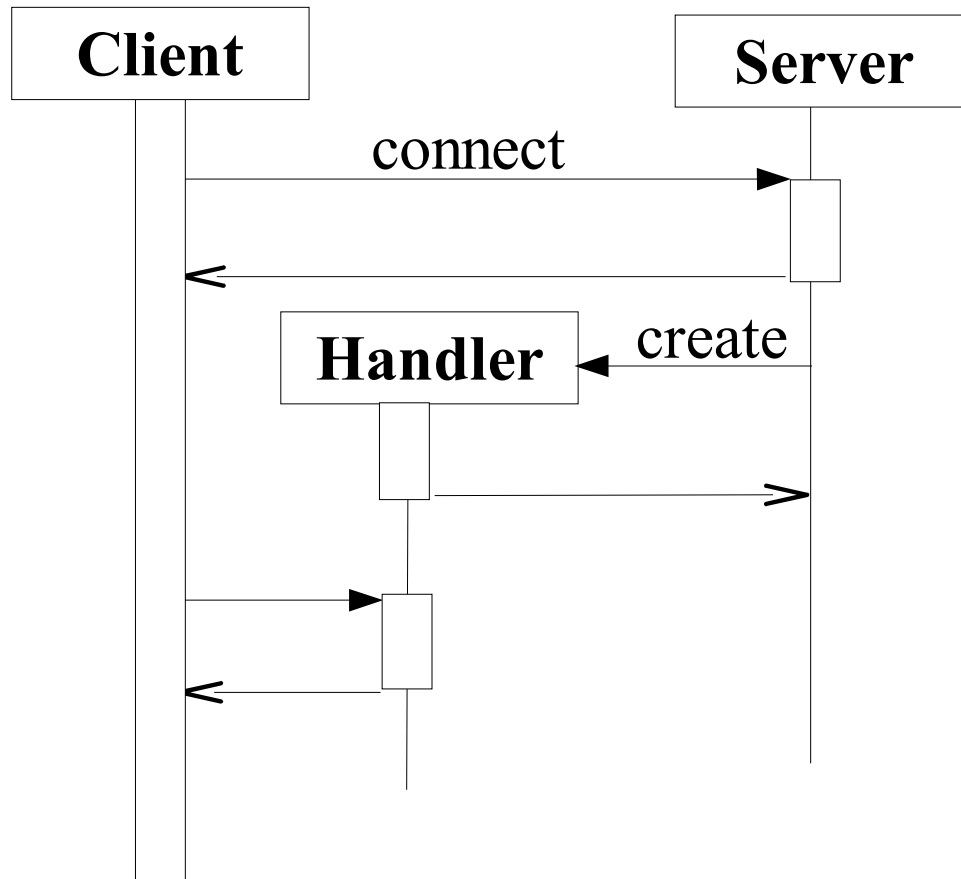
3



4



Server TCP multi-threaded



Server TCP multi-threaded

Creazione della porta di ascolto, quindi creazione e avvio di un nuovo gestore a fronte di ogni nuova richiesta proveniente da un client

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

Server TCP multi-threaded

Gestione della richiesta in un thread separato

```
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();  
}
```

MTEchoServer.Handler

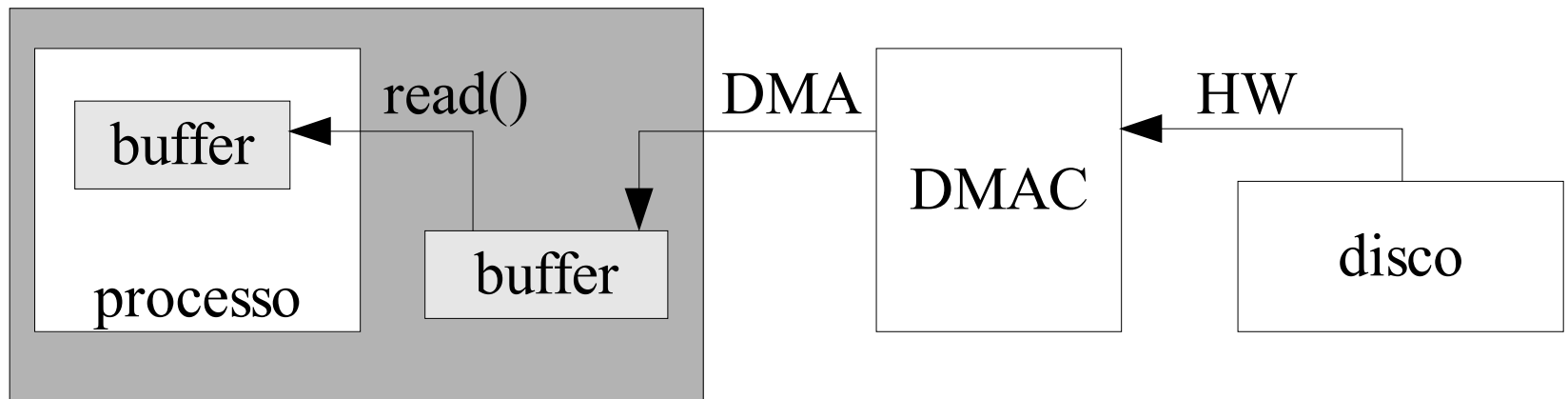
Server TCP con readiness selection

Java NIO è nuovo insieme di classi Java basate su un differente modello per I/O: **block** anziché **stream** I/O, che si beneficia di meccanismi specifici dei S.O.

- **Buffer**: contiene dati che devono essere scritti o sono stati letti
- **Channel**: trasferisce efficientemente dati tra un buffer e l'altro estremo (socket, file); a differenza di uno stream, può essere bidirezionale

Server TCP con readiness selection

Con l'introduzione dei Buffer si cercano di minimizzare i trasferimenti dati tra spazio utente e sistema



Server TCP con readiness selection

```
ByteBuffer buffer = ByteBuffer.wrap( MOTD.getBytes() );
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.socket().bind( new InetSocketAddress( port ) );
ssc.configureBlocking( false );
while( true ) {
    SocketChannel cc = ssc.accept();
    if ( null != cc ) {
        buffer.rewind();
        cc.write( buffer );
        cc.close();
    }
    else
        Thread.sleep( 5000 );
}
```

NIOMOTDServer

Server TCP con readiness selection

I/O multiplexing, o readiness

selection è un meccanismo tramite il quale, con un'unica chiamata di sistema si individuano tutti i e soli i canali pronti a svolgere operazioni di I/O

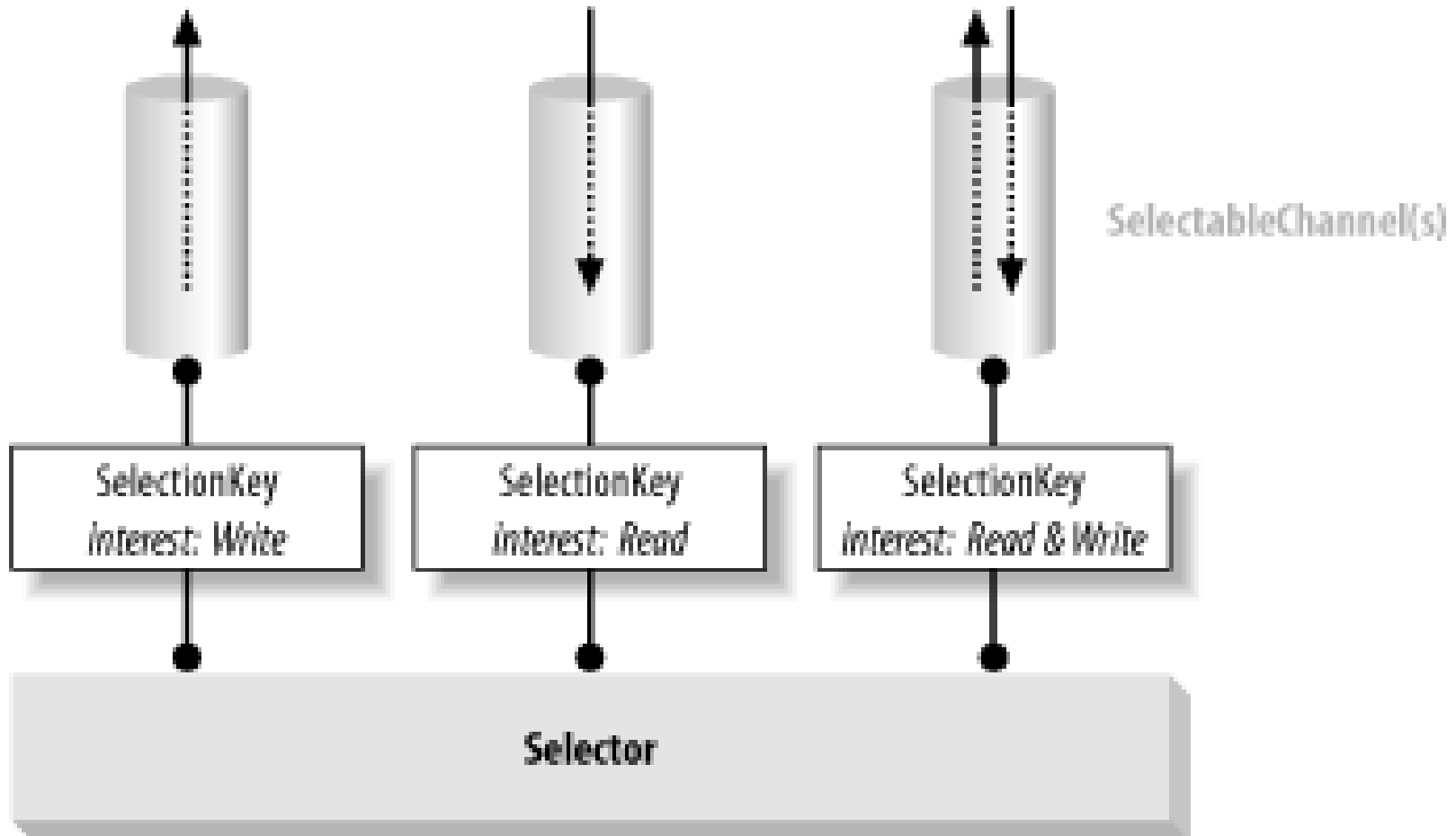
- Multiplexing di un gran numero di sockets aperte consente
 - operazioni in modalità non bloccante
 - singolo thread può gestire molte sockets
- Non comporta inefficienza del polling

Server TCP con readiness selection

Principio

- Oggetti di tipo `SelectableChannel` sono registrati con un `Selector`
 - `SelectionKey` codifica relazione canale-selettore
- Fra i canali registrati si seleziona l'insieme dei canali pronti
- La chiave di selezione indica le operazioni di interesse e quelle pronte
- Si ignorano canali inattivi (non così nel polling)

Server TCP con readiness selection



Server TCP con readiness selection

Creazione del canale per la porta di ascolto, del selettore e registrazione del canale sul selettore

```
ServerSocketChannel ssc = ServerSocketChannel.open();  
ssc.socket().bind( new InetSocketAddress( port ) );  
ssc.configureBlocking( false );  
Selector selector = Selector.open();  
ssc.register( selector, SelectionKey.OP_ACCEPT );
```

Server TCP con readiness selection

Ciclo principale: readiness selection ed eventuale gestione della comunicazione con i canali pronti

```
while( true ) {  
    selector.select();  
    Iterator it = selector.selectedKeys().iterator();  
    while( it.hasNext() ) {  
        SelectionKey key = (SelectionKey) it.next();  
        it.remove();  
        if( key.isAcceptable() )  
            accept(key);  
        if( key.isReadable() )  
            handle( key );  
    }  
}
```

Server TCP con readiness selection

Accettazione di una connessione entrante, e registrazione del nuovo canale per la lettura

```
public void accept( SelectionKey key, Selector selector ) {  
    ServerSocketChannel s = (ServerSocketChannel) key.channel();  
    SocketChannel cc = s.accept();  
    cc.configureBlocking( false );  
    cc.register( selector, SelectionKey.OP_READ );  
}
```

Server TCP con readiness selection

Gestione della comunicazione con un client

```
public void handle( SelectionKey key ) {  
    SocketChannel channel = (SocketChannel) key.channel();  
    buffer.clear();  
    int count;  
    while ( ( count = channel.read( buffer ) ) > 0 ) {  
        buffer.flip();  
        while ( buffer.hasRemaining() )  
            channel.write( buffer );  
        buffer.clear();  
    }  
    if ( count < 0 )  
        channel.close();  
}
```

NIOEchoServer

Server TCP con readiness selection

I/O multiplexing evita i costi associati alla creazione (solo in parte mitigabili con thread pools) e allo scheduling dei threads

Nei server le connessioni aperte possono essere migliaia, e la gestione dei threads può comprometterne il funzionamento

Può comunque essere utile anche in un client (es. browser web)

Server TCP con readiness selection

Il rovescio della medaglia: scrivere i programmi è più difficile!

Ad esempio, si deve gestire esplicitamente lo stato delle singole connessioni (mentre con i threads è implicito)

Reactor

Volendo fare ordine e formalizzare... c'è spazio per un altro pattern architetturale

Reactor consente ad applicazioni guidate da eventi di separare (demultiplex) e distribuire (dispatch) le richieste di servizio che sono inviate all'applicazione da uno o più clienti

Noto anche come **dispatcher**, **notifier**

Reactor

Problema: in un sistema distribuito, le applicazioni basate su eventi devono poter gestire molteplici richieste che pervengono simultaneamente, anche se sono elaborate in maniera sincrona e sequenziale. L'arrivo di ogni richiesta è identificato da un evento indication. L'applicazione deve quindi separare e smistare le richieste.

Reactor

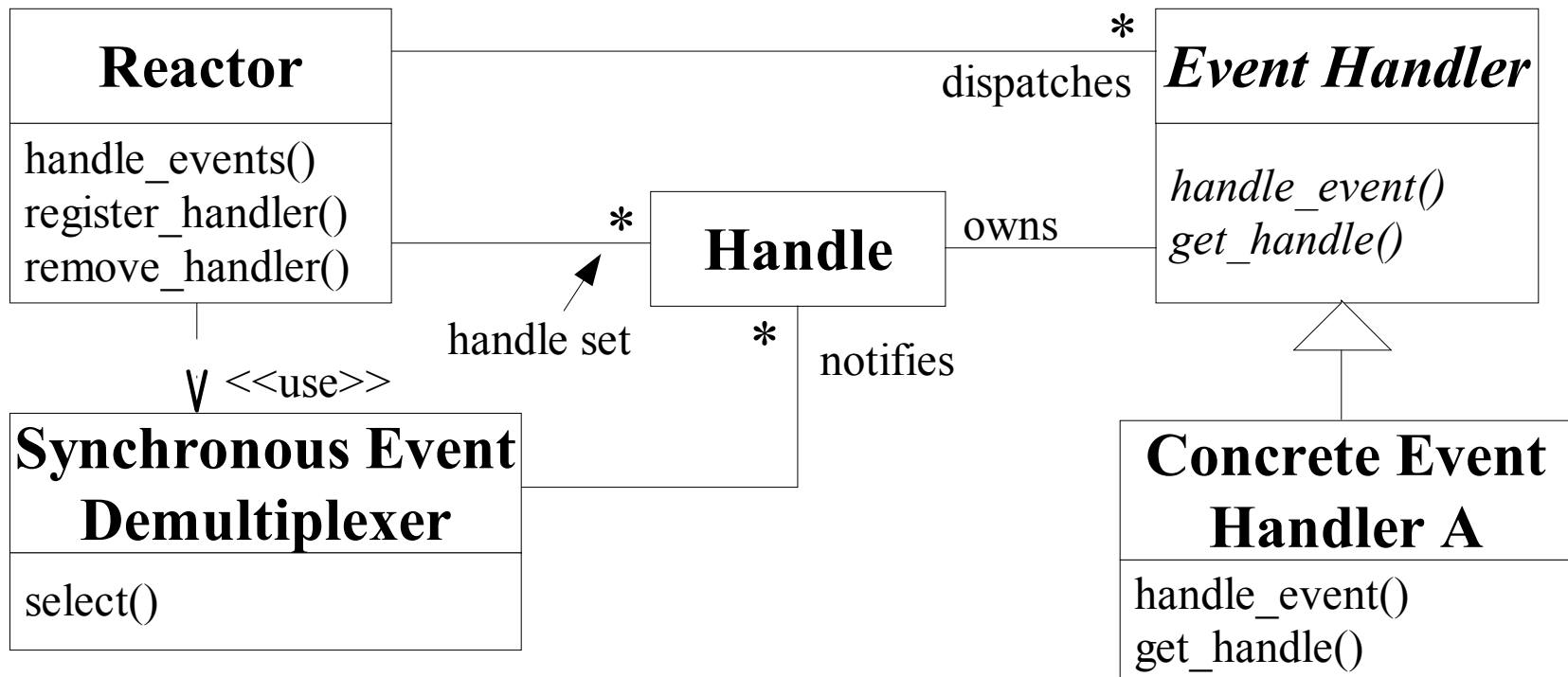
Forces

- migliorare prestazioni e latenza, non bloccando l'applicazione su una singola sorgente di eventi
- massimizzare la produttività
- integrazione di nuovi o migliorati servizi
- il codice applicativo dovrebbe essere liberato da problemi di multi-threading e sincronizzazione

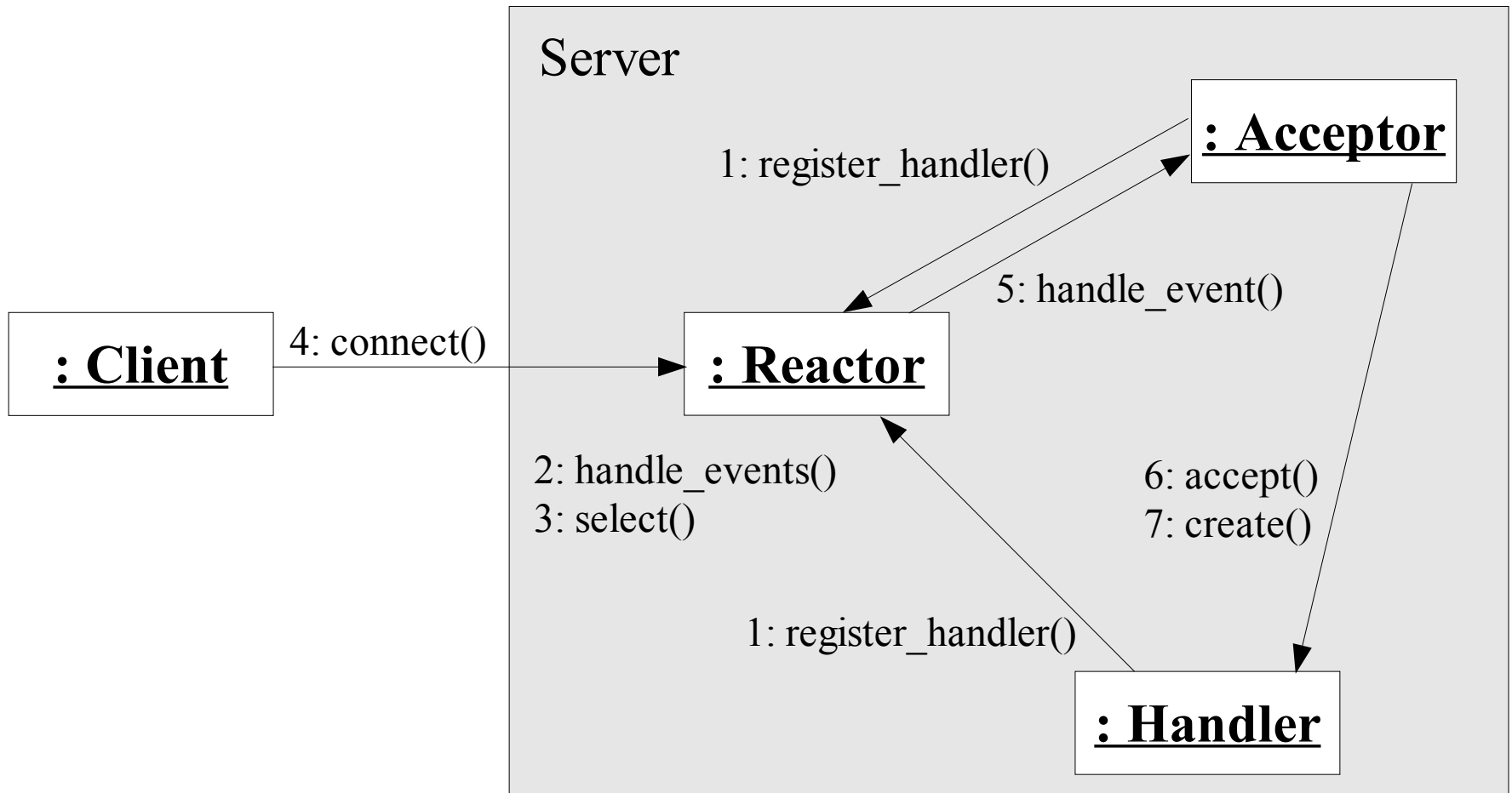
Reactor

Soluzione: attendere in maniera sincrona l'arrivo di eventi da una o più sorgenti. Integrare i meccanismi di demultiplexing e dispatching. Disaccoppiare questi meccanismi dall'elaborazione specifica dell'applicazione

Reactor



Reactor

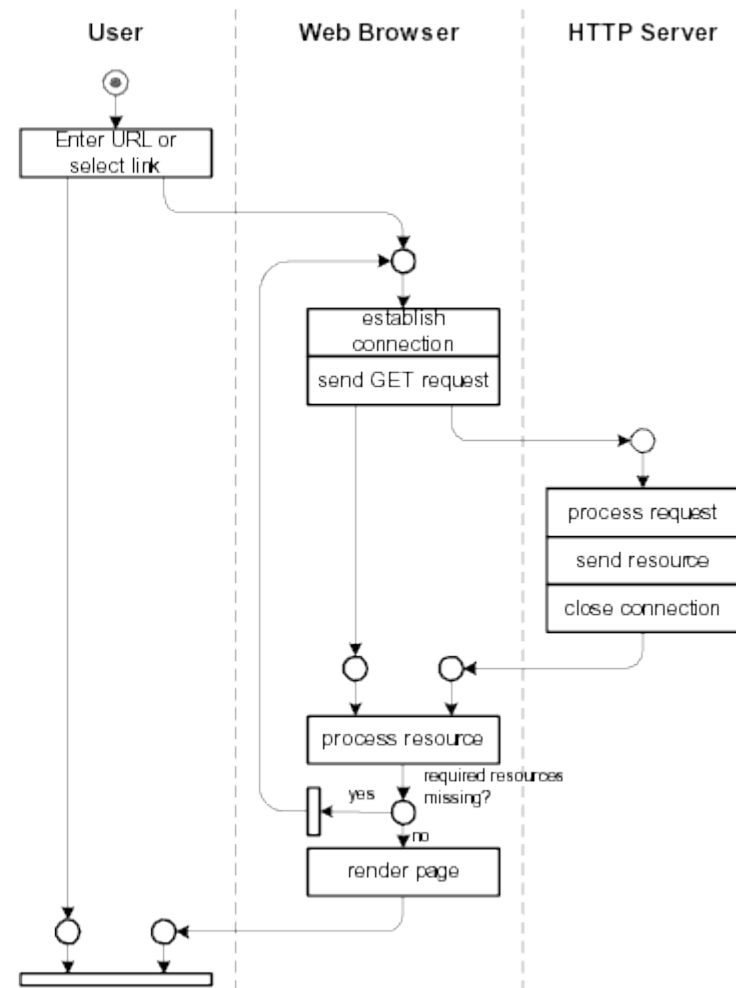
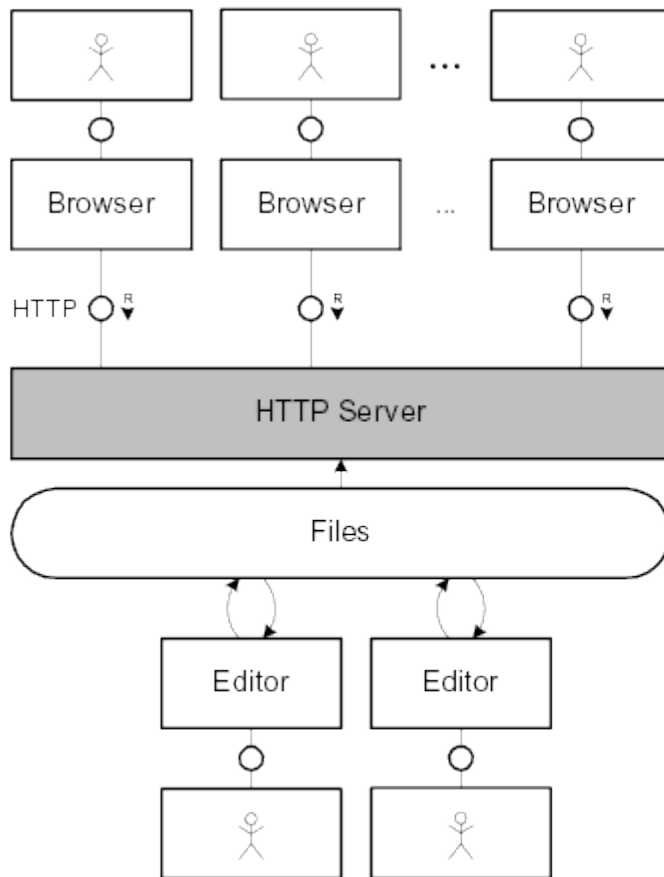


Apache MPM

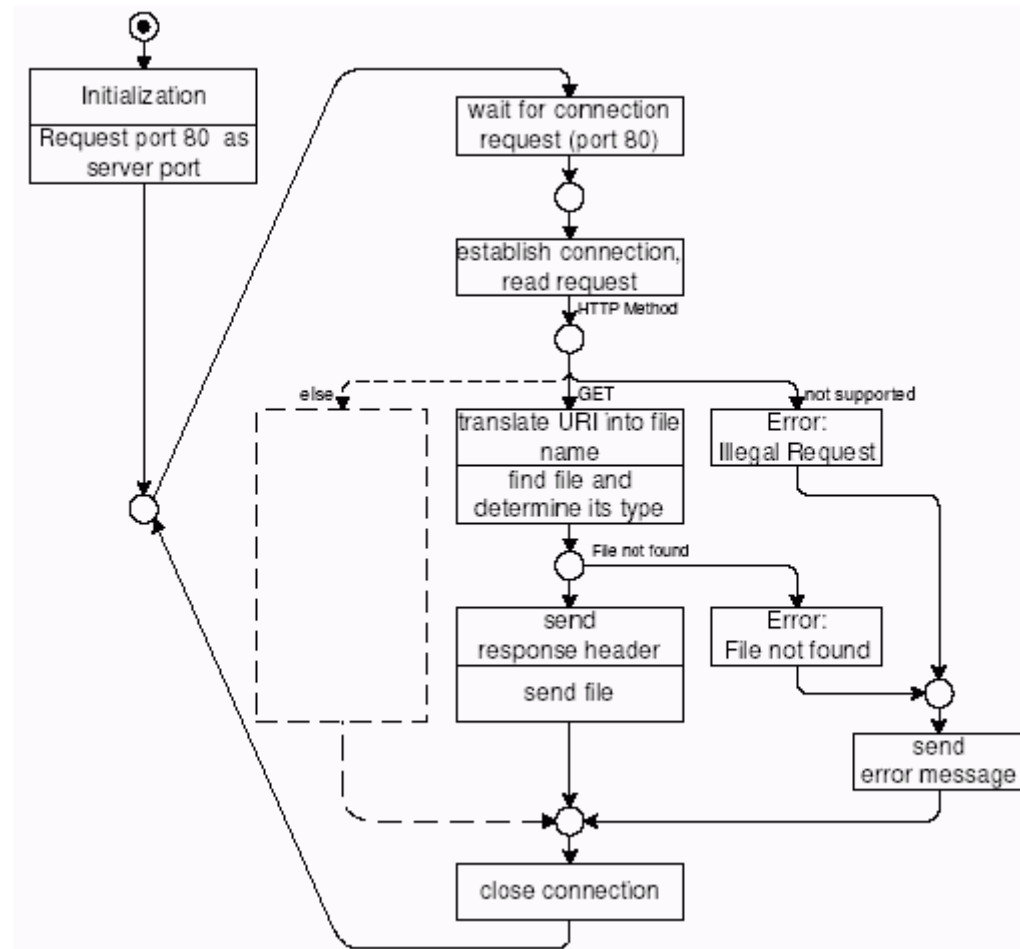
Il web server **Apache** è un buon **caso di studio** per confrontare diverse soluzioni architettureali, con particolare riferimento ai problemi di robustezza, prestazioni e scalabilità

Inoltre, da un lato è di interesse operativo per il **sistemista**, dall'altro è un modello di riferimento per lo **sviluppatore**

Apache MPM



Apache MPM



Apache MPM

La multiprogrammazione è indispensabile per un web server prestante

L'introduzione dei **Multi-Programming Modules** (MPM) è dovuta al fatto che ogni sistema operativo presenta caratteristiche diverse

certe architetture per la MP sono incompatibili o non sufficientemente performanti su determinati S.O., da cui l'esigenza di riconfigurabilità (statica) del web server

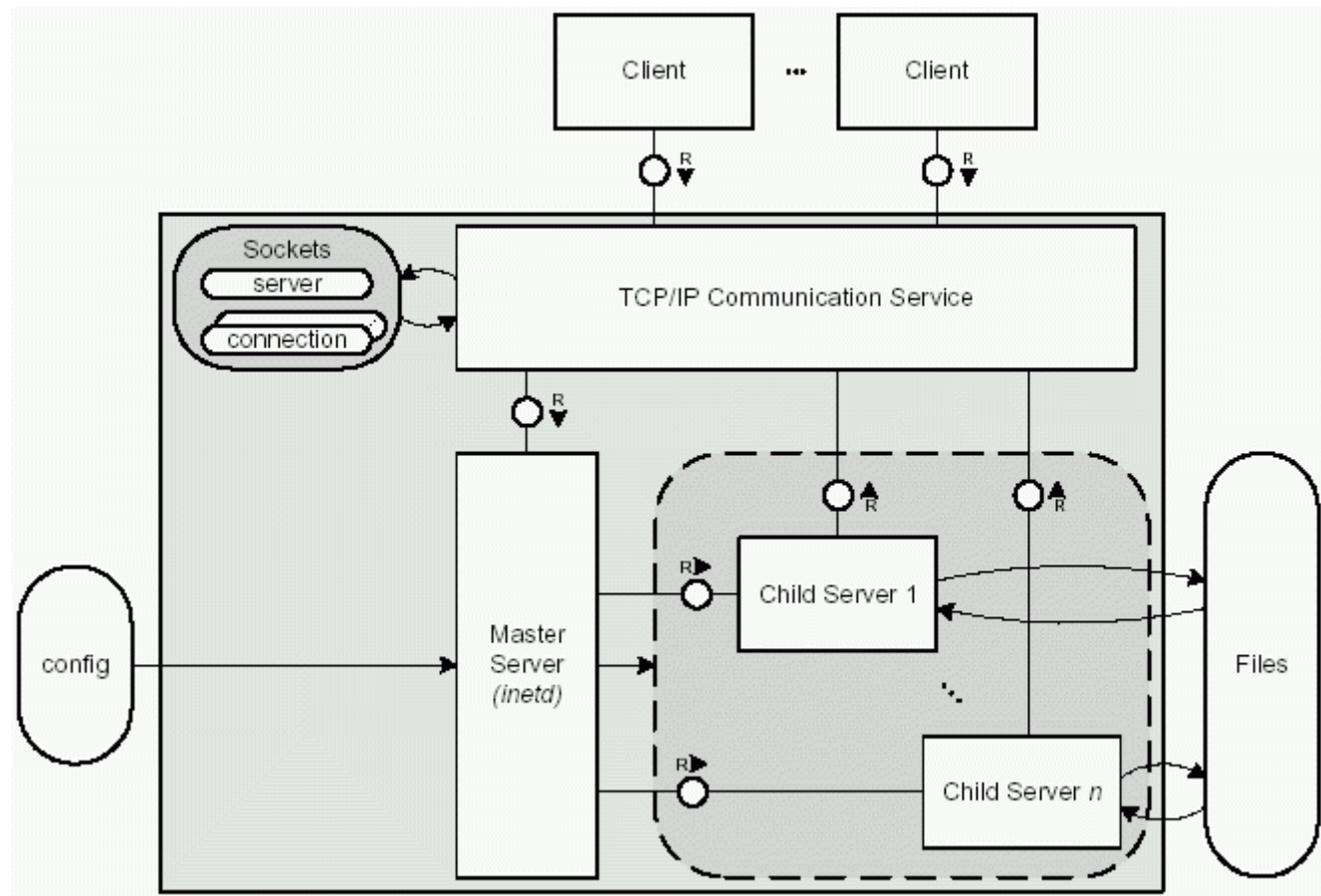
Apache MPM

Inetd (Unix/Linux) opera come “gatekeeper”:

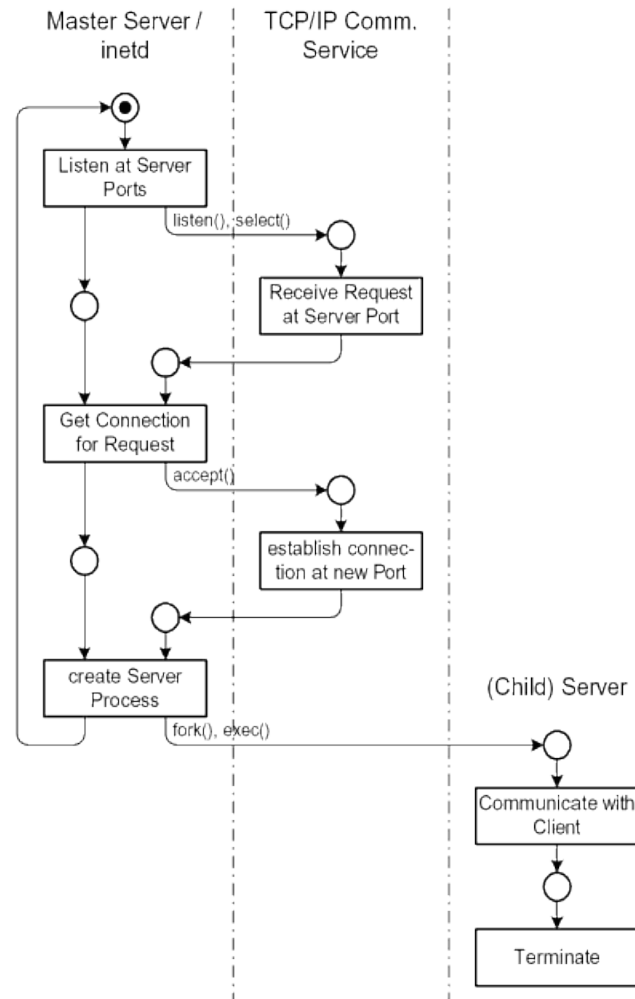
- è registrato per un servizio, ovvero è in ascolto sulla porta associata al servizio
- a fronte di una richiesta genera un nuovo processo tramite una `fork()`
- se il programma che deve gestire la richiesta è diverso dal master (come nel caso di Apache), si esegue anche una `exec()`

(v. anche design pattern Acceptor-Connector)

Apache MPM



Apache MPM



Apache MPM

- Utile se la gestione della richiesta dura nel tempo (cioè, la comunicazione non termina dopo la prima richiesta), e se in questo periodo deve essere mantenuta un'informazione di stato
- HTTP, però, è un protocollo senza stato, e la creazione di un nuovo processo richiede tempo, durante il quale il “portiere” non può ascoltare

Apache MPM

Architettura per la MP:

- come creare e controllare le unità operative (tasks)
- come comunicare le richieste da elaborare

Principali variabili di scelta:

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

Apache MPM

- Modalità di comunicazione tra tasks
- Sospensione dei tasks inattivi (idle)
 - Variabili di condizione, semafori, ...
- Ascolto delle connessioni entranti
 - Numero di listener
 - uno per ogni socket oppure uno per tutte le socket
 - Listener dedicato
 - Un listener e una coda dei job, oppure i task ricoprono sia il ruolo di listener che quello di worker

Apache MPM

MPM in Apache 1.3

- Unix: preforking
- WinNT: job que

MPM in Apache 2.0

- preforking
- WinNT
- worker
- leader
- per child

Apache MPM

L'architettura per la multiprogrammazione di Apache è basata sul concetto di **task pool**: all'avvio viene creato un insieme di unità operative, cui in seguito vengono delegate le richieste

- in fase di esecuzione non si incorre nei costi di generazione dei processi
- un processo **master server** controlla il pool

Apache MPM

Preforking

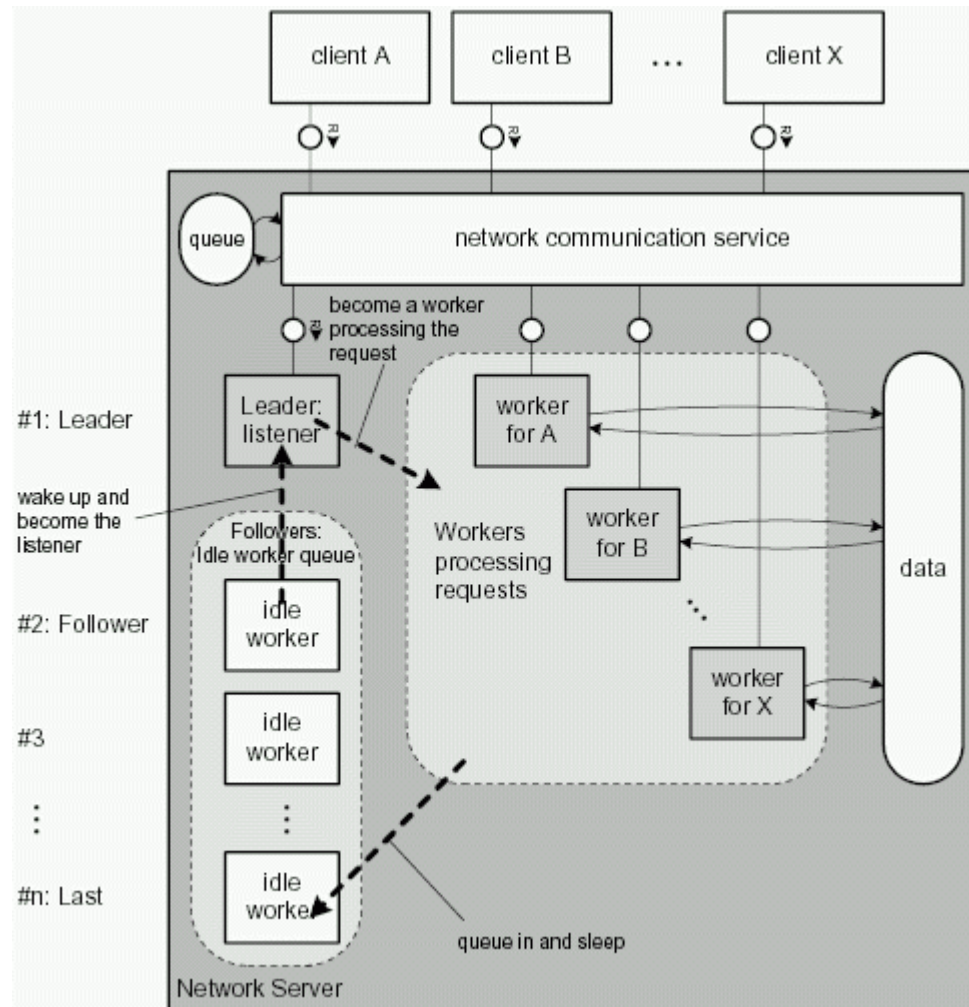
- all'avvio il master genera i processi figli
 - ogni figlio ha un solo thread
- a regime, il master controlla il numero di processi intattivi per determinare dinamicamente la dimensione del pool
- i figli si mettono in ascolto e gestiscono le richieste
 - l'accesso alla porta di ascolto avviene in mutua esclusione, secondo un pattern leader/follower

Apache MPM

Leader/follower [Schmidt et al., 2000]

- pool di tasks
- tasks svolgono i ruoli di listener e worker
 - non è richiesta comunicazione tra tasks
- esiste al più un listener (**leader**)
 - i task inattivi si accodano per poter divenire listener (**followers**)
- pipe of death
 - master deposita particolare job nella coda

Apache MPM

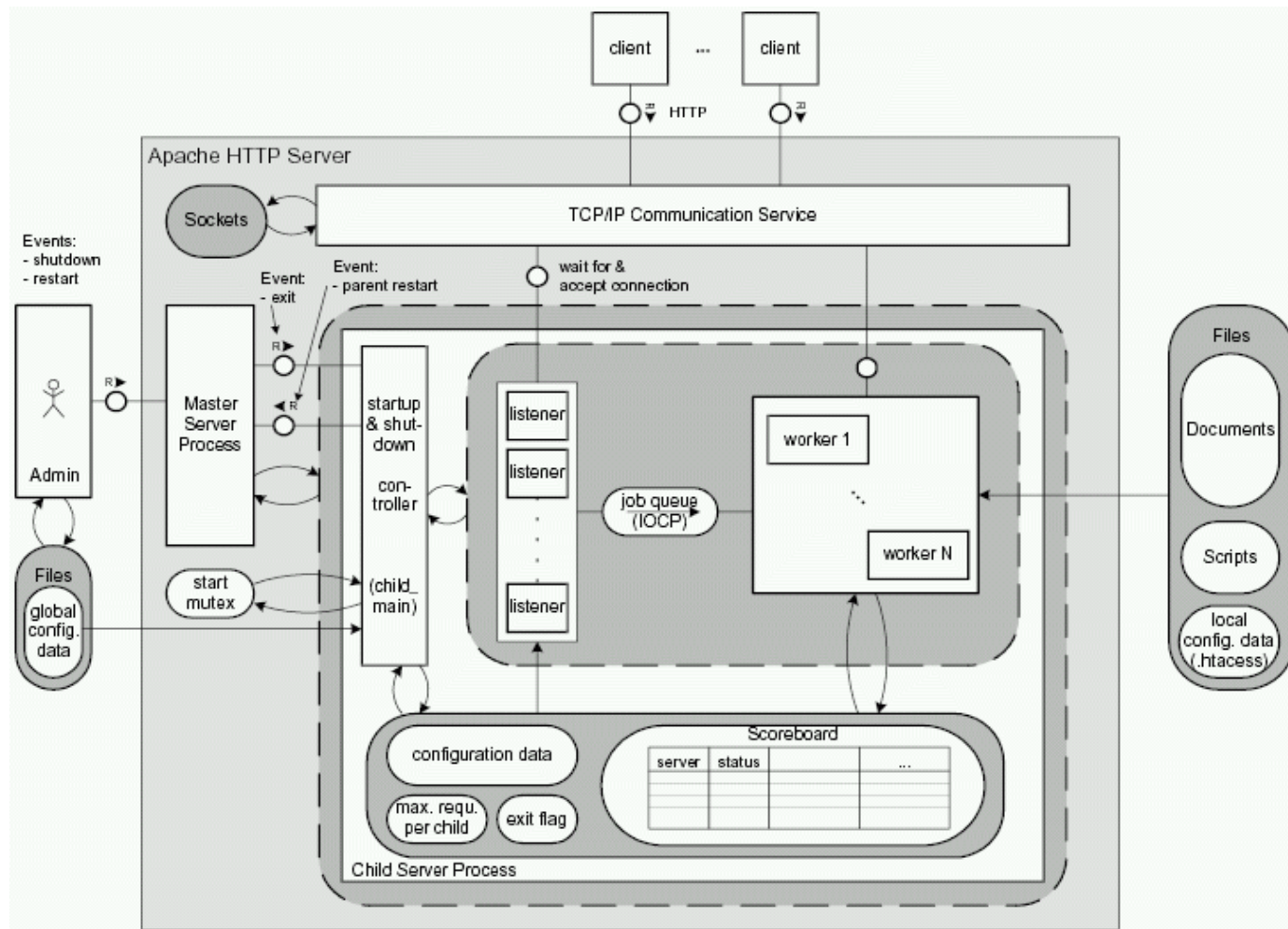


Apache MPM

WinNT

- threads anziché processi
- numero fisso di figli
 - i threads inattivi non influenzano le prestazioni
- 2 processi
 - supervisor
 - worker (tipi di threads: master, worker, listener)
- job queue in 1.3, I/O completion in 2.0
 - I/O completion consente limite superiore al numero di threads attivi

Apache MPM

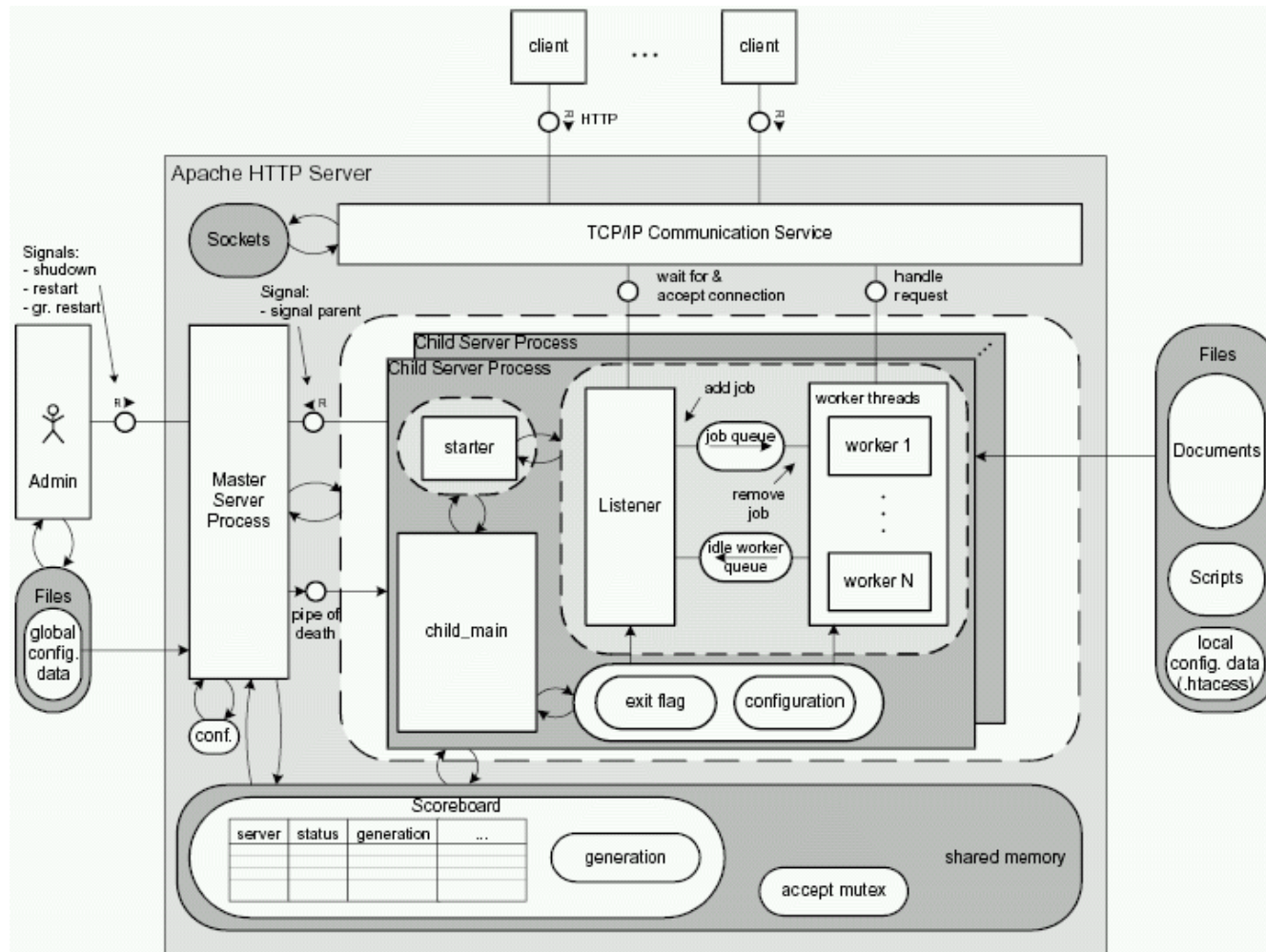


Apache MPM

Worker (per 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

Apache MPM



Apache MPM

Leader

variante di worker

- 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)

sperimentale

Apache MPM

Per child

simile a worker, ma

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

sperimentale

Apache MPM

Per saperne di più

- <http://httpd.apache.org/docs-2.0/>
- <http://apache.hpi.uni-potsdam.de/document>