

# SOCKET

## INTRODUZIONE: PROTOCOLLI DI TRASPORTO

Il livello di trasporto costituisce l'ultima parte di quello che possiamo definire **stack del protocollo di rete** di base, nel senso che implementa tutti quei servizi non forniti dall'interfaccia di livello di rete; il livello di trasporto trasforma infatti la rete sottostante in qualcosa di utilizzabile (canale logico) da parte degli sviluppatori di applicazioni. In particolare, l'idea è quella di offrire un servizio **affidabile** per la comunicazione tra processi attraverso la rete. Il protocollo **TCP (Tranfer Control Protocol)** risponde perfettamente a tali aspettative.

### Servizio TCP :

- *Orientato alla connessione:* il client invia al server una richiesta di connessione
- *Trasporto affidabile (reliable transfer)* tra processi mittente e ricevente
- *Controllo di flusso (flow control):* il mittente rallenta per non sommergere il ricevente
- *Controllo della congestione (congestion control):* il mittente rallenta quando la rete è sovraccarica
- *Non offre:* garanzie di banda e ritardo minimi

La comunicazione TCP/IP nel classico modello client server con lo **scambio di messaggi** attraverso le **socket**.

## INTERFACCIA SOCKET

**Osservazione:** come esempio, tratteremo **l'interfaccia socket** introdotta negli anni '70 in UNIX Berkeley.

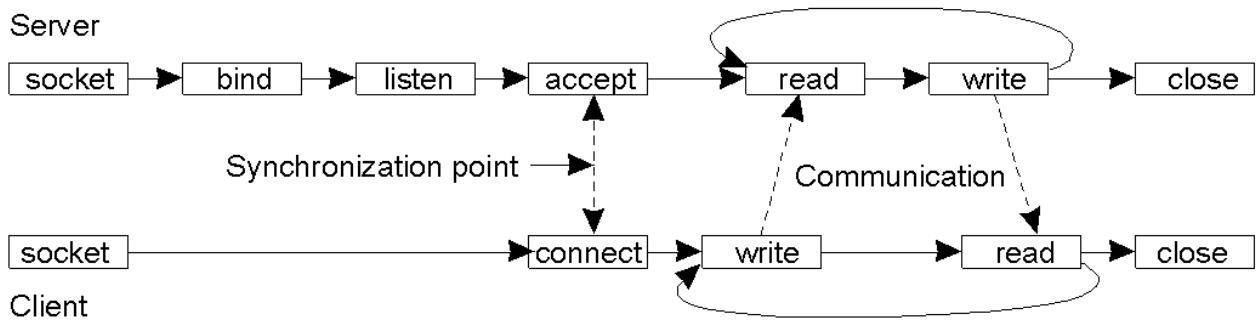
Concettualmente una **socket** è una porta di comunicazione su ciò un'applicazione può scrivere dati da inviare sulla rete sottostante e da cui poter leggere dati in ingresso. Una **socket** costituisce un'astrazione rispetto alla porta di comunicazione reale utilizzata dal sistema operativo locale per uno specifico protocollo di trasporto.

Nel seguito ci concentriamo sulle **primitive** per le socket del TCP, mostrate nella figura sottostante

Primitives	Meaning
<b>SOCKET</b>	Create a New Communication Endpoint.
<b>BIND</b>	Attach a Local Address to a SOCKET.
<b>LISTEN</b>	Shows the Willingness to Accept Connections.
<b>ACCEPT</b>	Block the Caller until a Connection Attempts Arrives.
<b>CONNECT</b>	Actively Attempt to Establish a Connection.
<b>SEND</b>	Send Some Data over Connection.
<b>RECEIVE</b>	Receive Some Data from the Connection.
<b>CLOSE</b>	Release the Connection.

In generale i **server** eseguono le prime quattro primitive, solitamente nell'ordine stabilito.

Quando richiama la primitiva **socket**, il chiamante crea una nuova porta di comunicazione per un protocollo di trasporto specifico. Internamente, creare una porta di comunicazione significa che il sistema operativo locale riserva delle risorse per gestire l'invio e la ricezione dei messaggi per il protocollo specificato. La primitiva **bind** associa un indirizzo locale con la socket appena creata. Il collegamento (*binding*) indica al sistema operativo che il server vuole ricevere messaggi solo all'indirizzo e sulla porta specificati. La primitiva **listen** viene richiamata solo nel caso di comunicazione **orientata alla connessione**. È una chiamata **non bloccante** che consente al SO di allocare un buffer per il numero di connessioni specificato che il server può supportare. Una chiamata alla **accept** blocca il chiamante (loop) finché non arriva una richiesta di connessione. Al suo arrivo, il sistema operativo locale crea una nuova socket con le stesse caratteristiche dell'originale e la restituisce al chiamante. Il server nel frattempo può tornare in ascolto sulla socket originale.



Vediamo ora il lato **client**; anche qui deve essere anzitutto creata una **socket** con la primitiva corrispondente, ma il *binding* esplicito della socket ad un indirizzo locale non è necessario (SO alloca dinamicamente la porta, *binding implicito*). La primitiva **connect** richiede che il chiamante specifichi l'indirizzo a livello di trasporto a cui va inviata la richiesta. Il client è quindi **bloccato** finché la connessione non è stabilita con successo.

A questo punto entrambi i lati possono iniziare a scambiarsi informazioni con le primitive **send** e **receive** (**read** e **write**). Infine la chiusura della connessione è simmetrica e si ha quando sia il client che il server richiamano la primitiva **close**.

# PROGETTARE APPLICAZIONI CON LE SOCKET

## CLIENT

Uno dei principali compiti del client è di fornire un mezzo all'utente per interagire con server remoti. Sono approssimativamente due i modi in cui questa interazione può essere supportata. Innanzitutto per ogni servizio remoto la macchina client avrà una controparte separata che può contattare il servizio di rete. Una seconda soluzione è di fornire accesso diretto ai servizi remoti unicamente un'interfaccia utente opportuna. Questo significa che la macchina client è usata solo come **terminale** senza bisogno di memoria logica. L'architettura risulta quindi concettualmente più semplice di quella di un server.

## SERVER

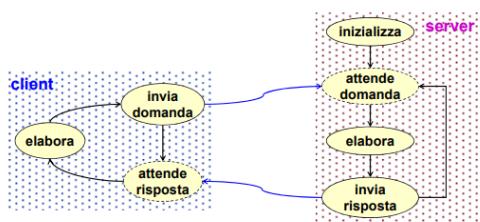
Un server è un processo che implementa un servizio specifico a nome di un insieme di client. In sostanza ogni server è organizzato nello stesso modo: attende una richiesta in ingresso da un client e successivamente assicura che la richiesta sia presa in carico, dopodiché attende la richiesta in ingresso successiva. Dunque in sintesi deve:

- Creare una **socket**;
- Assegnagli una porta nota;
- Entrare in un ciclo infinito in cui alternare:
  - Attesa richiesta
  - Soddisfa richiesta
  - Invia risposta

L' **affidabilità** di un server è strettamente dipendente dall'affidabilità della comunicazione tra lui e i suoi client. I modi di organizzare i server sono molti. In particolare un server può essere:

- **Server iterativo**, il server stesso gestisce la richiesta e, se necessario, restituisce una risposta al client;
- **Server concorrente**, il server non gestisce la richiesta in prima persona, ma la passa a un altro thread o processo, dopodiché si mette immediatamente in attesa di un'altra richiesta; i server **multi-thread** sono esempi di server concorrenti;

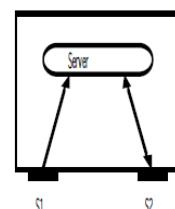
### Server iterativo



Al momento di una richiesta di connessione il server crea una socket temporanea per stabilire una connessione diretta con il client. Le eventuali ulteriori richieste per il server verranno accodate alla porta nota per essere successivamente soddisfatte.

Utilizzare un server iterativo comporta i seguenti **svantaggi**:

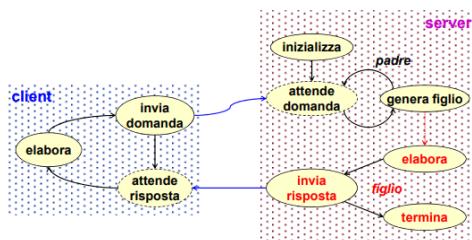
- Viene servito un client alla volta, gli altri devono attendere;
- Un server iterativo impedisce l'evoluzione di molti client;
- Bassa scalabilità, cresce il numero di utenti, diminuiscono le prestazioni;



Legenda:

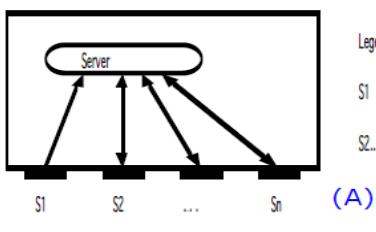
- S1 Socket per accettare richieste di connessione
- S2 Socket per connessioni individuali

## Server concorrente



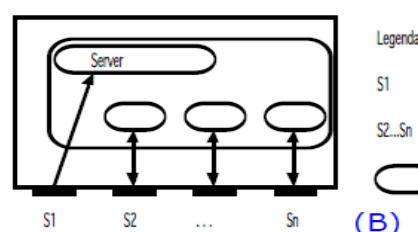
Un server concorrente può gestire più connessioni client. Per gestire le richieste attiva un sottoprocesso “figlio” e torna in attesa. La sua realizzazione può essere:

- **Simulata** con un solo processo (figura A); nel caso del linguaggio C si utilizza la funzione **select**, mentre per Java **Selector**, che restituiscono i canali **ready**;
- **Reale**, creando nuovi processi slave; nel caso del linguaggio C utilizzo la funzione **fork** (figura C), mentre per Java si introduce l'utilizzo dei **Thread** (figura B);

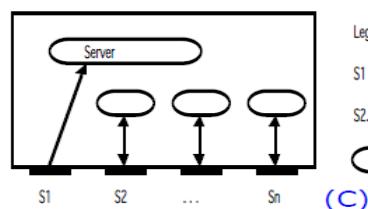


Legenda:  
S1  
S2...Sn

Socket per accettare  
richieste di connessione  
Socket per connessioni  
individuali



Legenda:  
S1  
S2...Sn  
Threads



Legenda:  
S1  
S2...Sn  
Processi slave

# HTML + DOM + CSS

## LINGUAGGI DI MURKUP

Quando si usa l'espressione **linguaggio di markup** ci si riferisce a un linguaggio che permette di descrivere (annotazioni) i dati attraverso una formattazione specifica che utilizza i cosiddetti tag, che non sono altro che dei marcatori.

Alcuni esempi di linguaggi di markup:

- TeX (e LaTeX)
- SGML
- **HTML**, XHTML, XML

Le annotazioni possono avere diverse finalità, come quella di **presentazione**, le quali definiscono come visualizzare il testo al quale sono associate; **procedurali**, definiscono istruzioni per programmi che elaborano il testo le quali sono associate; **descrittive**, etichettano semplicemente parti del testo, disaccoppiando la struttura dalla presentazione dal testo stesso.

## HTML

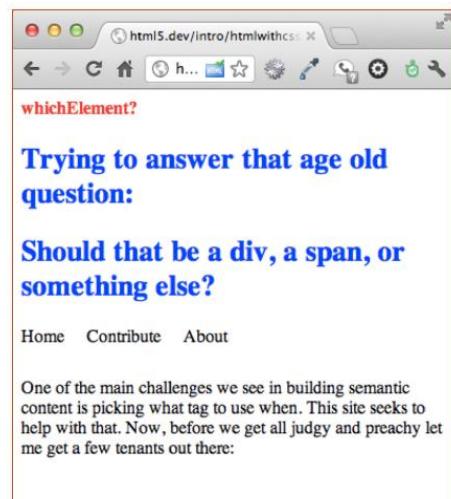
È un **linguaggio di markup** nato per la **formattazione e impaginazione di documenti ipertestuali** disponibili nel web, oggi è utilizzato principalmente per il **disaccoppiamento** della struttura logica di una pagina web (definita appunto dal markup) e la sua rappresentazione, gestita tramite gli stili **CSS**. La specifica HTML è definita dal W3C.

## CSS

Il **CSS (Cascading Style Sheets)** è un linguaggio usato per definire la formattazione di documenti HTML, XHTML e XML ad esempio i siti web e relative pagine web. Le regole per comporre il CSS sono contenute in un insieme di direttive emanate a partire dal 1996 dal W3C.

L'introduzione del CSS si è resa necessaria per **separare i contenuti delle pagine HTML dalla loro formattazione o layout** e permettere una programmazione più chiara e facile da utilizzare, sia per gli autori delle pagine stesse sia per gli utenti, garantendo contemporaneamente anche il riutilizzo di codice ed una sua più facile manutenzione.

```
<style type="text/css">
    h1{
        color: red;
        font-size: 16px;
    }
    h2{
        color: blue;
        font-size: 26px;
    }
    ul{
        list-style: none;
        clear: both;
        display: block;
        width: 100%;
        height: 30px;
        padding: 0;
    }
    li{
        float: left;
        padding-right: 20px;
    }
</style>
```



Per identificare e formattare particolari elementi o classi di elementi è possibile utilizzare i **selectors**;

```
p.intro {  
    color: red;  
}
```

Questo stile sarà applicato solo ai tag di classe intro

```
<p class="intro">
```

I principali **selettori** sono:

- **tag name**: il semplice nome del tag
  - p{...} //affects to all <p> tags
- **dot (.)**: applicabile a un tag, indica una classe
  - p.highlight{...} //affects all <p> tags with class="highlight"
- **sharp character (#)**: applicabile a un tag, indica un identificativo
  - p#intro{...} //affects to the <p> tag with the id="intro"
- **two dots (:)**: stati comportamentali (ad esempio evento mouseover)
  - p:hover{...} //affects to <p> tags with the mouse over
- **brackets ([attr='value'])**: tag con un valore specifico per un attributo 'value'
  - input[type="text"] {...} // affects to the input tags of the type text

## CSS: MEDIA QUERY

Le **media query** possono essere viste come particolari **selettori** capaci di valutare la capacità del device di accesso alla pagina; con queste ultime possiamo controllare ad esempio la **larghezza** e **altezza** del device o della finestra e l'**orientamento** dello schermo, la **risoluzione** ...

**Esempio:** se la pagina è più larga di 480 pixel (e si sta visualizzando sullo schermo) applica determinati stili agli elementi con id "*leftsidebar*" (un menu) e "*main*" (la colonna centrale)

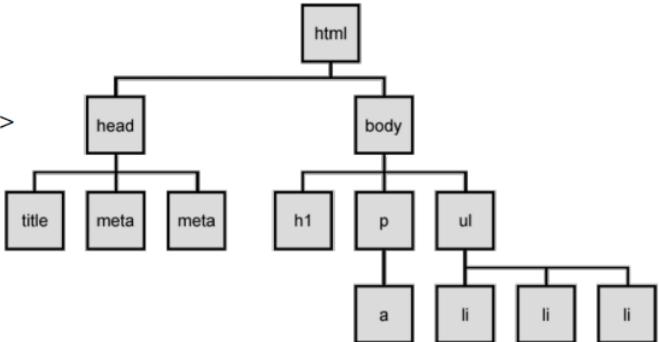
```
@media screen and (min-width: 480px){  
    #leftsidebar {width: 200px; float: left;}  
    #main {margin-left:216px;}  
}
```

**Osservazione:** l'impostazione di HTML e CSS separa nettamente il contenuto dalla modalità di visualizzazione; il concetto di "**separation of concerns**" è un principio di design altamente desiderabile in contesto informatico;

DOM

**DOM (Document Object Model)** è un'interfaccia neutrale rispetto al linguaggio di programmazione e alla piattaforma utilizzata per consentire ai programmi l'accesso e la modifica dinamica di contenuto, struttura e stile di un documento web. **DOM** è una **API** definita dal W3C e implementata da ogni browser moderno per l'accesso e la gestione dinamica dei documenti XML e HTML.

```
<html>
  <head>
    <title>Example ...</title>
    <meta charset="UTF-8">
    <meta name="author" content="John Doe">
  </head>
  <body>
    <h1>Heading 1</h1>
    <p>This is a paragraph of text with a
    <a href="/path/page.html">link in it</a>.
    </p>
    <ul>
      <li>Coffee</li>
      <li>Tea</li>
      <li>Milk</li>
    </ul>
  </body>
</html>
```



Ogni nodo può essere caratterizzato da attributi che ne facilitano l'identificazione, la ricerca, la selezione; questi possono essere:

- Un **identificatore univoco**
  - Una **classe** che indica l'appartenenza ad un insieme che ci è utile definire

Il browser stesso fornisce funzionalità di ricerca:

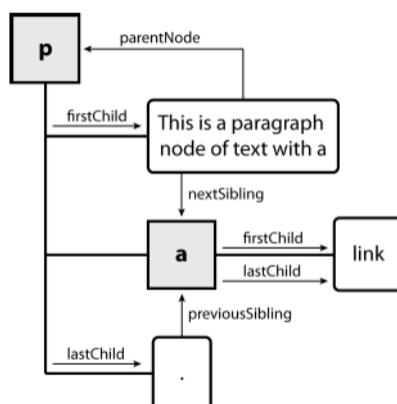
- `getElementById(IdName)`
  - `getElementsByClassName(Classname)`
  - .... altre ....

```
<h1>A One Page FAQ</h1>
<div id="introText">
  <p>bla bla bla</p>
</div>
```

```
<h2>I've heard that JavaScript is the  
long-lost fountain of youth. Is this  
true?</h2>  
<div class="answer">  
  <p>bla bla bla</p>  
</div>
```

```
<h2>Can JavaScript really solve all of  
my problems?</h2>  
<div class="answer">  
  <bla bla bla</p>  
</div>
```

```
<p id="foo">  
    This is a paragraph of text with a  
    <a href="/path/to/page.html">link</a>  
.  
</p>
```



## HTML 5: NOVITA'

- ❖ Nuovi **elementi semantic**i per la strutturazione delle pagine
  - article, section, aside, header, footer, etc ...
- ❖ Nuovi **elementi di input e multimediali**
  - Widget per input search, email, url, number, tel, ma anche range, date...
  - audio, video, canvas
- ❖ Nuove **API JavaScript** per la manutenzione delle pagine
  - Offline data, drag and drop, web storage

# HTTP

## PANORAMICA E DEFINIZIONI

**HTTP (hypertext transfer protocol)**, protocollo a **livello di applicazione** del Web, definito in [RFC 1945] e in [RFC 2616], costituisce il cuore del Web. Questo protocollo è implementato in due programmi, **client** e **server**, in esecuzione su sistemi periferici diversi che comunicano tra loro scambiandosi messaggi HTTP. Il protocollo definisce sia la struttura dei messaggi sia la modalità con cui client e server si scambiano i messaggi.



Il client è realizzato da un **Browser Web**, un programma che consente la navigazione del web da parte di un utente. La funzione primaria di un browser è quella di **interpretare il codice** con cui sono espresse le informazioni (**pagine web**) e visualizzarlo in forma di **ipertesto**.

Una **pagina web** Una pagina web (web page), detta anche documento, è costituita da oggetti. Un oggetto è semplicemente un file (quale un file HTML, un'immagine JPEG, un'applet Java, una clip video e così via) indirizzabile tramite un URL. La maggioranza delle pagine web consiste di un file HTML principale e diversi oggetti referenziati da esso.



Un **web server**, che implementa il lato server di HTTP, ospita oggetti web, indirizzabili tramite **URL**. Tra i più popolari ricordiamo *Apache* e *Microsoft Internet Information Server*. Un Web Server è sempre attivo, ha un indirizzo IP fisso e risponde potenzialmente alle richieste provenienti da milioni di diversi browser.

Un **ipertesto (hypertext)** è un insieme di testi o pagine leggibili con l'ausilio di un'interfaccia elettronica, in maniera non sequenziale, tramite hyperlink (o più semplicemente link, cioè collegamenti), che costituiscono una rete raggiata o variamente incrociata di informazioni organizzate secondo criteri paritetici o gerarchici (es. menu).

L' **URL** Identifica un oggetto nella rete e specifica come interpretare i dati ricevuti attraverso il **protocollo**. È composto da 5 componenti principali:

1. Nome del protocollo
2. Indirizzo dell'host
3. Ports del processo (la controparte)
4. Percorso dell'host
5. Identificatore di risorsa

**protocollo://indirizzo\_IP[:porta]/cammino/risorsa**

1.	2.	3.	4.	5.
----	----	----	----	----

## Esempi:

- <ftp://www.adobe.com/download/acroread.exe>
- <http://www.biblio.unimib.it/go/Home/Home-English/Services>
- <http://www.biblio.unimib.it/link/page.jsp?id=47502837>
- <http://www.someSchool.edu/someDept/pic.gif>
- <http://www.someSchool.edu:80/someDept/pic.gif>

HTTP utilizza **TCP** (anziché UDP) come protocollo di trasporto, il quale mette a disposizione di http un servizio di trasferimento affidabile: http non si deve preoccupare dei dati smarriti o di come TCP recuperi le perdite o riordini i dati all'interno della rete.

È importante notare che il server invia i file richiesti ai client senza memorizzare alcuna informazione di stato a proposito del client. Dato che i server HTTP non mantengono informazioni sui client, HTTP è classificato come un **protocollo senza memoria di stato** (*stateless protocol*).

## FORMATO DEI MESSAGGI HTTP

Le specifiche HTTP [RFC 1945; RFC 2616; RFC 7540] includono la definizione dei due formati dei messaggi HTTP, di richiesta e di risposta.

### Messaggio di richiesta HTTP

Ecco un tipico **messaggio chi richiesta** http

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

Notiamo innanzitutto che il messaggio è scritto in testo **ASCII**, in modo che l'utente sia in grado di leggerlo. Inoltre, notiamo che consiste di cinque righe, ciascuna seguita da un carattere di ritorno a capo (*carriage return*) e un carattere di nuova linea (*line feed*). L'ultima riga è seguita da una coppia di caratteri di ritorno a capo e nuova linea aggiuntivi.

La prima riga è detta **riga di richiesta (request line)** e quelle successive **righe di intestazione (header lines)**. La riga di richiesta presenta tre campi: il campo **metodo**, il campo **URL** e il campo **versione di HTTP**. La riga *Host: www.someschool.edu* specifica **l'host** su cui risiede l'oggetto. Si potrebbe pensare che questa riga di intestazione non sia necessaria, dato che è già in corso una connessione TCP con l'host. Includendo la linea di intestazione *Connection: close*, il browser sta comunicando al server che non si deve occupare di connessioni persistenti, ma vuole che questi chiuda la connessione dopo aver inviato l'oggetto richiesto. La riga di intestazione *User-agent*: specifica il tipo di browser che sta effettuando la richiesta al server, in questo caso *Mozilla/5.0*, un browser Firefox. Questa riga è utile in quanto il server può inviare versioni diverse dello stesso oggetto a browser di tipi diversi. Ciascuna delle versioni viene indirizzata dallo stesso URL. Infine, *Accept-language*: indica che l'utente preferisce ricevere una versione in francese dell'oggetto se disponibile; altrimenti, il server dovrebbe inviare la versione di default. La riga *Accept-language*: rappresenta solo una delle molte intestazioni di negoziazione dei contenuti disponibili in HTTP.

## METODI

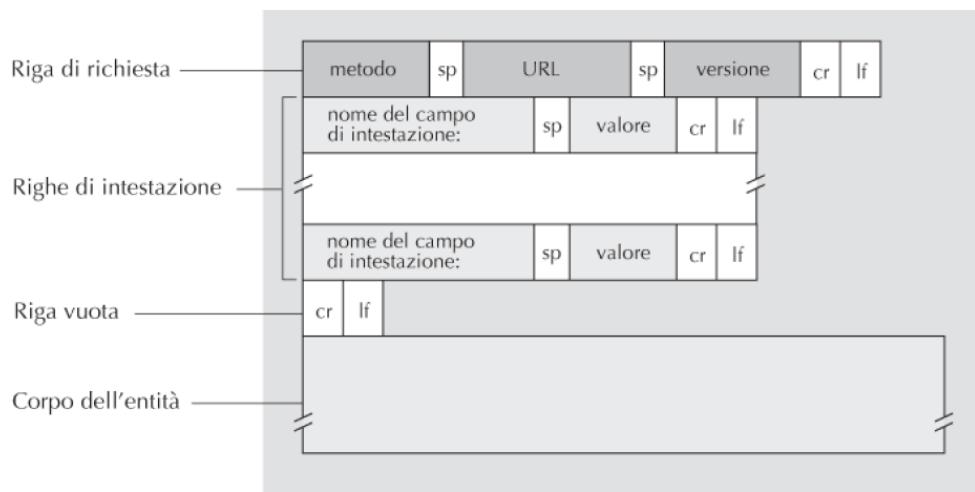
- Metodo **GET**: utilizzato dal client per ottenere un documento dal server, include un eventuale input in coda alla URL della risorsa; restituisce una rappresentazione della risorsa. Un uso tipico del metodo può essere per richiedere pagine html e immagini;
- Metodo **POST**: comunica dei dati da elaborare lato server o crea una nuova risorsa subordinata all'URL indicata. Viene utilizzato per esempio per processare una form e modificare dati in un DB;
- Metodo **HEAD**: simile al metodo GET ma viene restituito solo l'Head della pagina Web; è spesso utilizzato in fase di debugging;
- Metodo **PUT**: è utilizzato per memorizzare un documento nel server nella posizione specificata dall' URL;
- Metodo **DELETE**: cancella il documento specificato nella URL;
- Metodo **TRACE**: traccia una richiesta, visualizzando come viene trattata dal server;

**Osservazione:** il metodo GET si dice **safe**, ovvero l'esecuzione non ha effetti sul server infatti si tratta di un metodo per la sola lettura; la risposta può essere gestita con una **cache** dal client;

**Osservazione:** il metodo POST non è **idempotente**, ovvero ogni esecuzione ha un diverso effetto, al contrario per esempio del metodo DELETE (più DELETE si comportano come un'unica richiesta);

	cache	safe	idempotent
OPTIONS			✓
GET	✓	✓	
HEAD	✓	✓	
POST			
PUT			✓
DELETE			✓
TRACE			✓

A questo punto concentriamoci sul formato generale di un messaggio di richiesta (Figura 2.8). Notiamo che questo segue da vicino l'esempio precedente. dopo le linee di intestazione (e i caratteri di ritorno a capo e di nuova linea) si trova un “**corpo**” (*entity body*). Quest'ultimo è vuoto nel caso del metodo GET, ma viene utilizzato dal metodo POST.



**Figura 2.8** Formato generale dei messaggi di richiesta di HTTP.

## Messaggi di risposta HTTP

Presentiamo ora un tipico messaggio di risposta HTTP che potrebbe rappresentare la risposta al messaggio di richiesta dell'esempio precedente.

```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html
(data data data data data...)
```

Analizzando in dettaglio questo messaggio di risposta, osserviamo tre sezioni: una **riga di stato** iniziale, sei **righe di intestazione** e il **corpo**. Quest'ultimo è il fulcro del messaggio: contiene l'oggetto richiesto (rappresentato da: data data data data data...). La **riga di stato** presenta tre campi: la versione del protocollo, un codice di stato e un corrispettivo messaggio di stato. In questo esempio, la riga di stato indica che il server sta usando [HTTP/1.1](#) e che tutto va bene (ossia che il server ha trovato e sta inviando l'oggetto richiesto).

Osserviamo ora le **righe di intestazione**. Il server utilizza la riga di intestazione *Connection: close* per comunicare al client che ha intenzione di chiudere la connessione TCP dopo l'invio del messaggio. La riga *Date*: indica l'ora e la data di creazione e invio, da parte del server, della risposta HTTP.

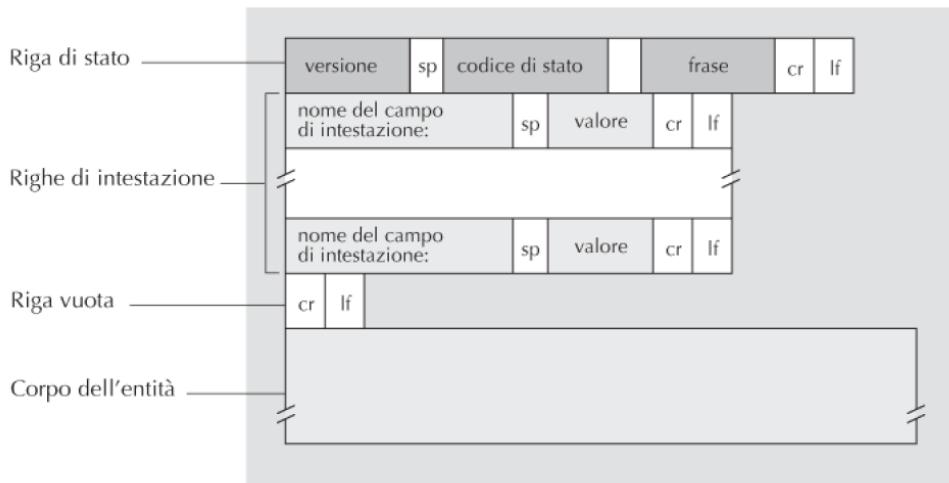
**Osservazione:** non si tratta dell'istante in cui l'oggetto è stato creato o modificato per l'ultima volta, ma del momento in cui il server recupera l'oggetto dal proprio file system, lo inserisce nel messaggio di risposta e invia il messaggio.

La riga *Server*: indica che il messaggio è stato generato da un web server Apache; essa è analoga alla riga User-agent: nel messaggio di richiesta HTTP. La riga *Last-Modified*: indica l'istante e la data il cui l'oggetto è stato creato o modificato per l'ultima volta. Tale riga è importante per la gestione dell'oggetto nelle cache, sia nel client locale sia in alcuni server in rete (*proxy server o proxy*). La riga di intestazione *Content-Length*: contiene il numero di byte dell'oggetto inviato. La riga *Content-Type*: indica che l'oggetto nel corpo è testo HTML. Il tipo dell'oggetto viene ufficialmente identificato tramite l'intestazione Content-Type: e non tramite l'estensione del file.

### Note:

- [Client HTTP 1.0](#): Server chiude connessione al termine della richiesta;
- [Client HTTP 1.1](#): mantiene aperta la connessione oppure chiude se la richiesta contiene Connection: close;

Dopo l'esempio esaminiamo il formato generale di un messaggio di risposta (Figura 2.9) che rispecchia il precedente esempio.



**Figura 2.9** Formato generale dei messaggi di risposta di HTTP.

## CODICI DI STATO

Spediamo qualche parola aggiuntiva sui codici di stato e sulle loro espressioni. Il **codice di stato** e **l'espressione associata** indicano il **risultato della richiesta**. Tra i più comuni codici di stato e relative espressioni troviamo:

- **200 OK**: la richiesta ha avuto successo e in risposta si invia l'informazione;
- **301 Moved Permanently**: l'oggetto richiesto è stato trasferito in modo permanente; il nuovo URL è specificato nell'intestazione `Location:` del messaggio di risposta. Il client recupererà automaticamente il nuovo URL;
- **400 Bad request**: si tratta di un codice di errore generico che indica che la richiesta non è stata compresa dal server;
- **404 Not found**: il documento richiesto non esiste sul server;
- **505 Version Not Supported**: il server non dispone della versione di protocollo HTTP Richiesta;

**Nota:** più in generale i codici vengono raggruppati nelle seguenti categorie:

- **1xx** - Messaggio informativo. Questi codici di stato sono indicativi di una risposta temporanea. ...
- **2xx** - Esito positivo. ...
- **3xx** - Reindirizzamento. ...
- **4xx** - Errore del client. ...
- **5xx** - Errore del server. ...

## GET CONDIZIONALE

Sebbene il **web caching** riduca i tempi di risposta percepiti dall'utente, introduce un nuovo problema: **la copia di un oggetto che risiede in cache potrebbe essere scaduta**. Fortunatamente, HTTP presenta un meccanismo che permette alla cache di verificare se i suoi oggetti sono aggiornati. Questo meccanismo è chiamato **GET condizionale** (*conditional GET*). Un messaggio di richiesta HTTP viene detto messaggio di GET condizionale se

- 1) usa il metodo GET
- 2) include una riga di intestazione *If-modified-since*:

**Esempio:** Per prima cosa un proxy invia un messaggio di richiesta a un web server per conto del browser richiedente:

*GET /fruit/kiwi.gif HTTP/1.1*

*Host: www.exotiquecuisine.com*

Poi, il web server invia al proxy un messaggio di risposta con l'oggetto richiesto:

*HTTP/1.1 200 OK*

*Date: Sat, 3 Oct 2015 15:39:29*

*Server: Apache/1.3.0 (Unix)*

*Last-Modified: Wed, 9 Sep 2015 09:23:24*

*Content-Type: image/gif*

*(data data data data data...)*

Il proxy inoltra l'oggetto al browser richiedente e pone anche l'oggetto nella cache locale. Va sottolineato che la cache memorizza con l'oggetto anche la data di ultima modifica. Poi, una settimana più tardi, un altro browser richiede lo stesso oggetto attraverso il proxy, e l'oggetto si trova ancora nella cache. Dato che tale oggetto può essere stato modificato nel web server durante la settimana trascorsa, il proxy effettua un controllo di aggiornamento inviando un **GET condizionale**. Più nello specifico invia:

*GET /fruit/kiwi.gif HTTP/1.1*

*Host: www.exotiquecuisine.com*

*If-modified-since: Wed, 9 Sep 2015 09:23:24*

Si osservi che il valore della riga di intestazione *If-modified-since*: equivale esattamente al valore della riga di intestazione *Last-Modified*: inviata dal server una settimana prima. Questo GET condizionale sta comunicando al server di inviare l'oggetto solo se è stato modificato rispetto alla data specificata. Supponiamo che l'oggetto non sia stato modificato dalle 9:23:24 del 9 settembre 2015. Allora il web server invia un messaggio di risposta al proxy:

*HTTP/1.1 304 Not Modified*

*Date: Sat, 10 Oct 2015 15:39:29*

*Server: Apache/1.3.0 (Unix)*

*(corpo vuoto)*

Notiamo che in risposta a un GET condizionale, il web server invia ancora un messaggio di risposta, ma **non include l'oggetto richiesto**, in quanto ciò implicherebbe solo spreco di banda e incrementerebbe il tempo di risposta percepito dall'utente, in particolare se l'oggetto è grande. La riga di stato **304 Not Modified** comunica al proxy che può procedere e inoltrare al browser richiedente la copia dell'oggetto presente in cache.

## COOKIE

Abbiamo precedentemente visto che i server HTTP sono privi di stato. Ciò semplifica la progettazione e consente di sviluppare web server ad alte prestazioni, in grado di gestire migliaia di connessioni TCP simultanee. Tuttavia, è spesso auspicabile che i web server possano autenticare gli utenti, sia per limitare l'accesso da parte di questi ultimi sia per fornire contenuti in funzione della loro identità.

A questo scopo, HTTP adotta i **cookie**. I cookie, definiti in [RFC 6265], consentono ai server di tener traccia degli utenti. La maggior parte dei siti commerciali usa i cookie. La tecnologia dei cookie presenta quattro componenti:

1. Una riga di intestazione nel messaggio di risposta http
2. Una riga di intestazione del messaggio di richiesta http
3. un file mantenuto sul sistema dell'utente e gestito dal browser
4. un database sul sito.

Un esempio di utilizzo dei cookie è mostrato nella seguente figura

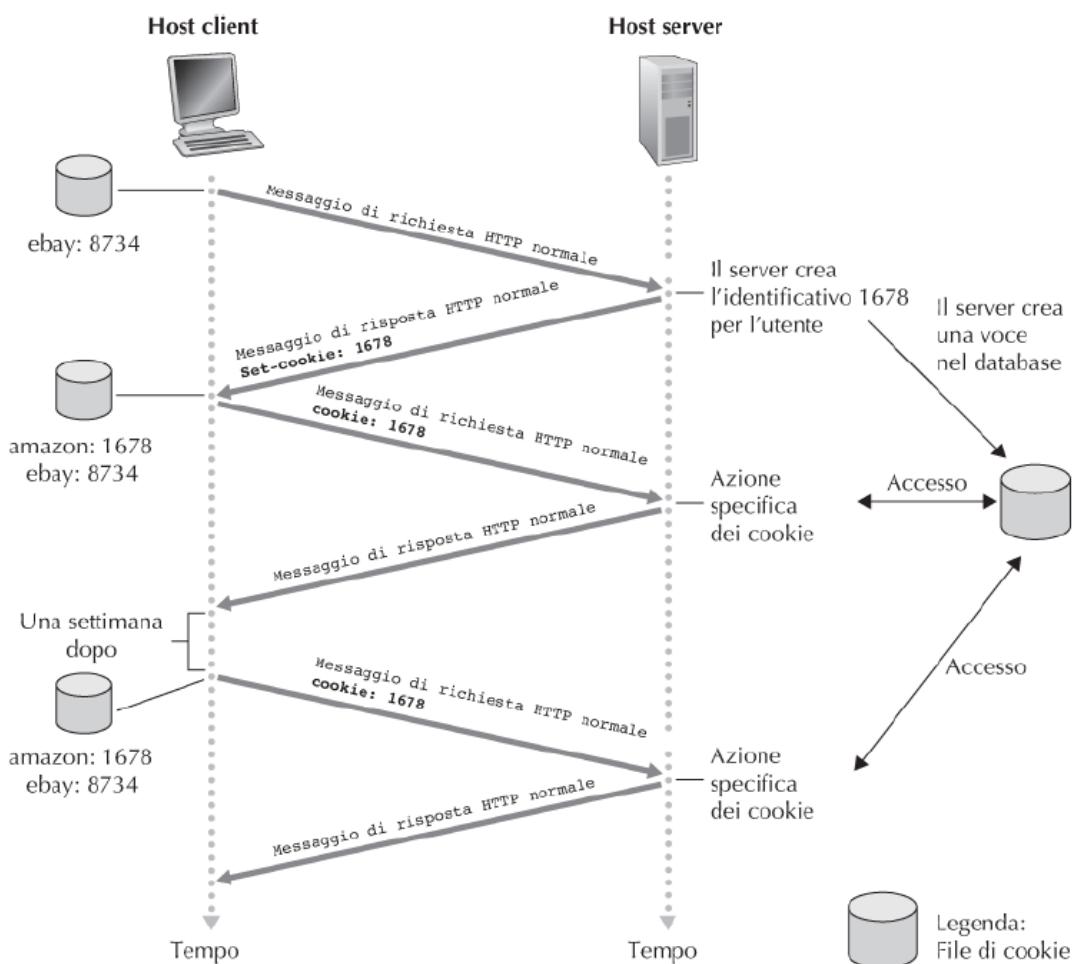
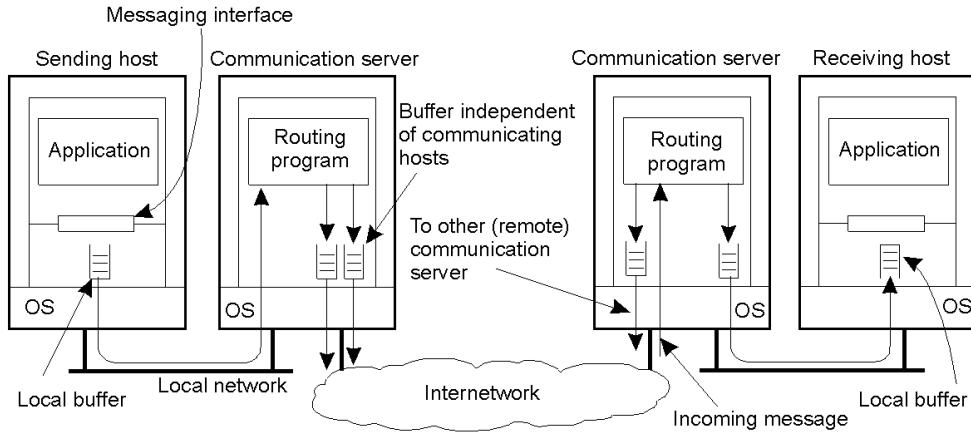


Figura 2.10 Memorizzazione dello stato dell'utente con i cookie.

## MESSAGE ORIENTED COMMUNICATION

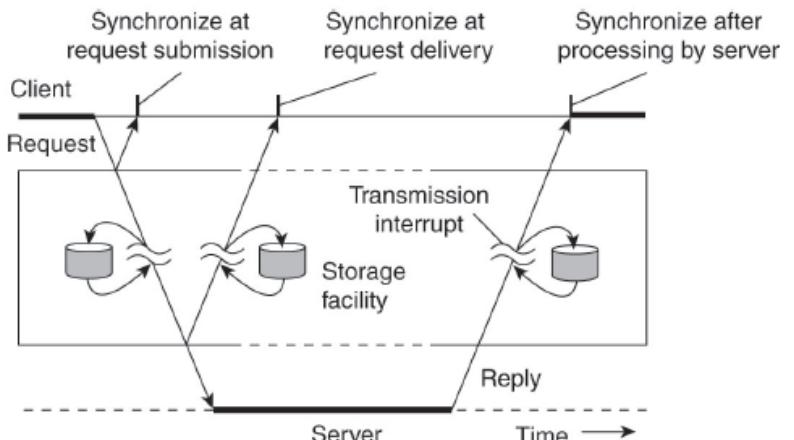
La comunicazione tra processi è il cuore di tutti i sistemi distribuiti. La comunicazione di questi ultimi si basa sempre sullo **scambio di messaggi** a basso livello come fornito dalla rete sottostante. Nel seguito analizzeremo un modello di comunicazione middleware orientati ai messaggi: **MOM(MO Middleware)**.



### TIPI DI COMUNICAZIONE

Con la **comunicazione persistente** un messaggio immesso per essere trasmesso viene memorizzato dal middleware per tutto il tempo che gli serve per consegnarlo al destinatario. In questo caso, il middleware memorizzerà il messaggio su uno o più elementi per la memorizzazione come mostrato in figura.

Di conseguenza non è necessario che l'applicazione mittente continui l'esecuzione dopo aver sottomesso il messaggio, e analogamente non è necessario che il mittente sia in esecuzione quando viene sottomesso il messaggio. Un classico esempio di **comunicazione persistente** è l'**E-mail**.

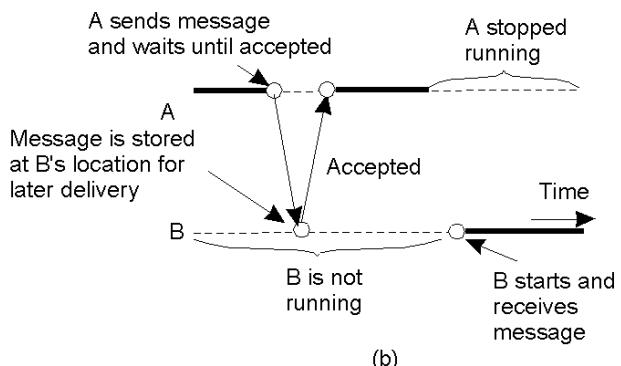
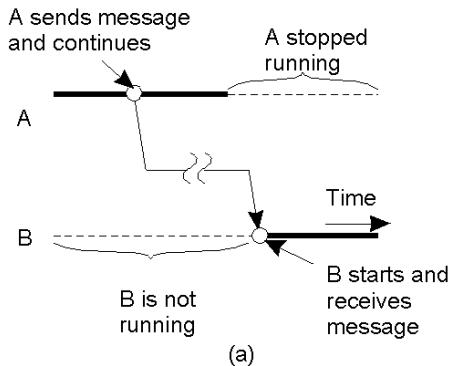


Diversamente con la **comunicazione transiente** un messaggio viene memorizzato dal sistema di comunicazione solo finché le applicazioni mittente e destinataria sono in esecuzione. Più precisamente, se il middleware non può consegnare un messaggio esso sarà semplicemente scaricato.

In questo caso il sistema di comunicazione consiste di **router** tradizionali memorizza e invia (*store and forward*).

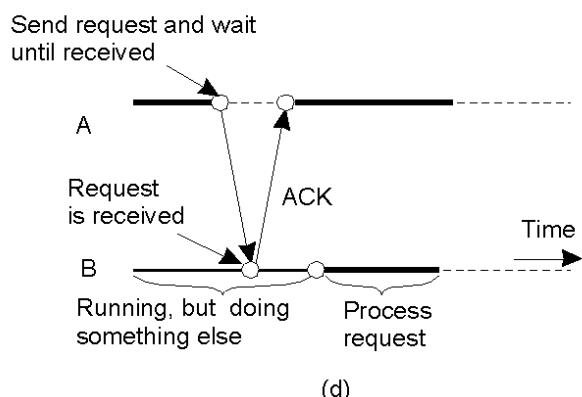
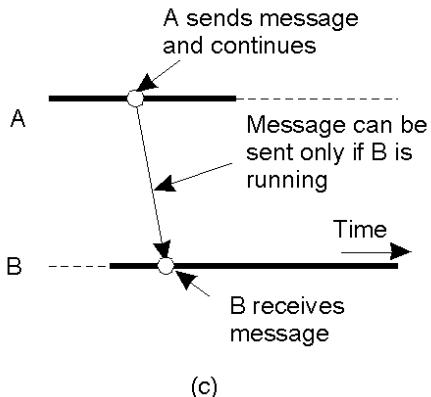
Oltre a queste ultime due caratteristiche, la comunicazione può essere **sincrona** o **asincrona**. La caratteristica della **comunicazione asincrona** è che un mittente continua l'elaborazione subito dopo aver sottomesso il suo messaggio. Questo significa che il messaggio viene (temporaneamente) memorizzato dal middleware immediatamente dopo la sottomissione. Con la **comunicazione sincrona**, il mittente è bloccato finché la sua richiesta non viene accettata.

## COMBINAZIONI TRA TIPI DI COMUNICAZIONE



a) **Persistente asincrona**

b) **Persistente sincrona**



c) **Transiente asincrona**

d) **Transiente sincrona**

----- o ----- o -----

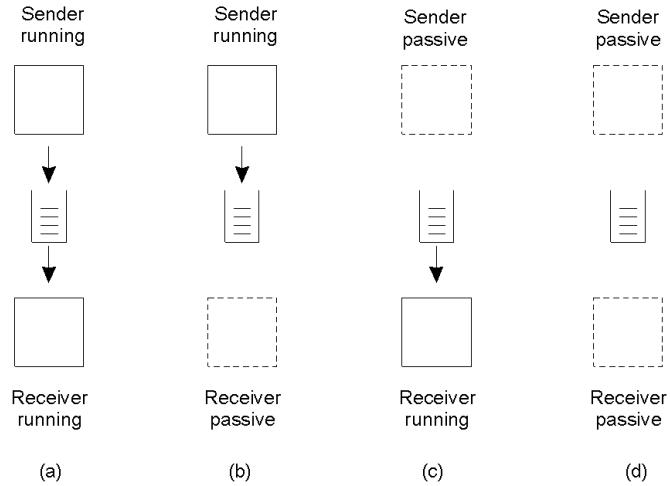
## SISTEMI A CODE DI MESSAGGI

I **sistemi a code di messaggi** forniscono un ampio supporto alla comunicazione **asincrona persistente**. Il punto essenziale di questi sistemi è che offrono la possibilità di memorizzare i messaggi a medio termine, senza che il mittente e il destinatario siano attivi durante la trasmissione del messaggio.

L'idea alla base di un sistema a code per lo scambio di messaggi è che le applicazioni comunicano inserendo dei messaggi in code specifiche. Questi messaggi vengono inoltrati attraverso server di comunicazione al fine di consegnarli al destinatario, anche se quest'ultimo non era attivo al momento dell'invio del messaggio.

Un aspetto importante dei sistemi a code di messaggi è che un mittente ha in genere solo la garanzia che il suo messaggio verrà alla fine inserito nella coda del destinatario, ma nessuna garanzia che quest'ultimo verrà realmente letto.

Non è necessario che mittente e destinatario siano in esecuzione nei momenti in cui i messaggi vengono immessi nelle rispettive code (l'immagine mostra le 4 possibili combinazioni):



In linea teorica i messaggi possono contenere qualsiasi dato, possono essere però inseriti delle restrizioni sulla **dimensione**.

### INTERFACCIA PER LE CODE

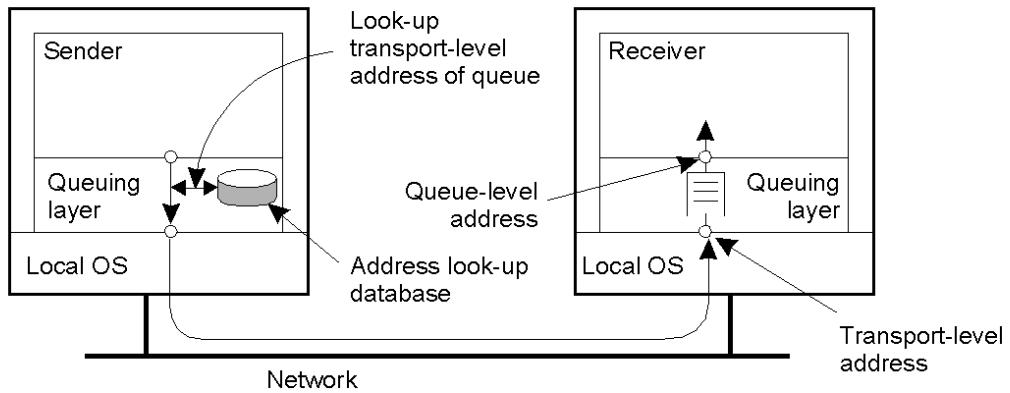
L'interfaccia di base fornita alle applicazioni è estremamente semplice;

Primitive	Meaning
<b>Put</b>	<b>Append a message to a specified queue</b>
<b>Get</b>	<b>Block until the specified queue is nonempty, and remove the first message</b>
<b>Poll</b>	<b>Check a specified queue for messages, and remove the first. Never block.</b>
<b>Notify</b>	<b>Install a handler to be called when a message is put into the specified queue.</b>

La primitiva **put** viene chiamata da un mittente per passare al sistema sottostante un messaggio da aggiungere alla coda specificata (chiamata non bloccante). La primitiva **get** è una chiamata bloccante attraverso la quale un processo autorizzato può rimuovere dalla coda specificata il messaggio pendente da più tempo. Il processo si blocca solo se la coda è vuota. La variante non bloccante è fornita dalla primitiva **poll**. Se la coda è vuota o un messaggio non può essere trovato il processo semplicemente va avanti.

Molti sistemi di code permettono di installare **handler** sotto forma di una funzione di **callback**, che prelevano in automatico i messaggi dalla coda quando un processo è in esecuzione.

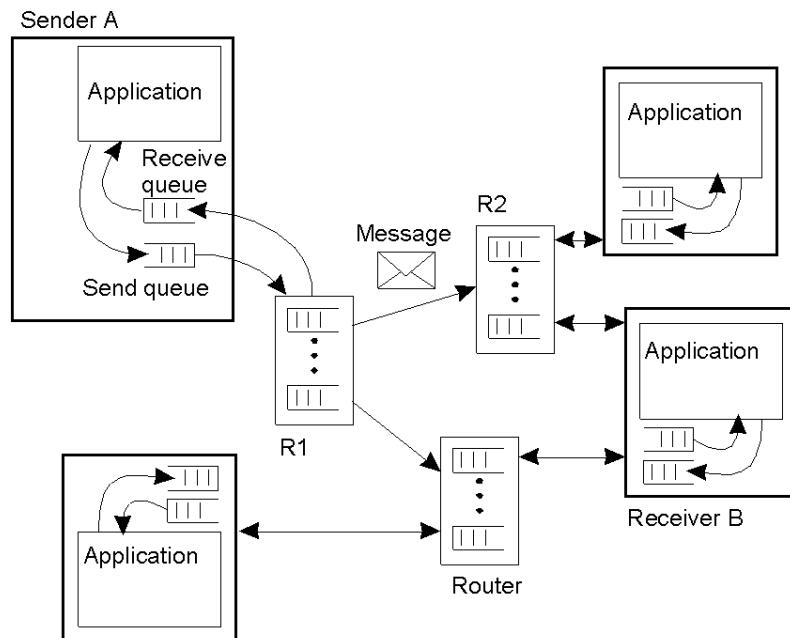
## ARCHITETTURA GENERALE DI UN SISTEMA DI CODE PER LO SCAMBIO DI MESSAGGI



L'insieme di code è distribuito su molte macchine, di conseguenza, per poter trasferire i messaggi un sistema deve mantenere una corrispondenza tra le code e le loro posizioni di rete. In pratica, questo significa che deve mantenere una base di dati (eventualmente distribuita) di **nomi delle code** e delle rispettive posizioni sulla rete, come mostrato nella figura sopra (da notare l'analogia con le corrispondenze DNS).

Le code sono gestite dai **gestori delle code**. Un **gestore** interagisce direttamente con l'applicazione che sta inviando o ricevendo un messaggio. Tuttavia esistono gestori speciali che agiscono da **router** o **relay**: essi inoltrano messaggi in ingresso ad altri gestori. In questo modo un sistema a code può gradualmente crescere fino a diventare una **rete overlay** completa a livello applicativo, basata su una rete di computer esistente.

## ARCHITETTURA GENERALE DI UN SISTEMA DI CODE (ROUTER)



Quando un mittente A inserisce nella sua coda locale un messaggio per il destinatario B, questo messaggio è innanzitutto trasferito al router più vicino, diciamo **R1**, come mostrato nella figura. A questo punto il router in questione sa che cosa deve fare del messaggio e lo inoltra in direzione di B. per esempio, **R1** potrebbe

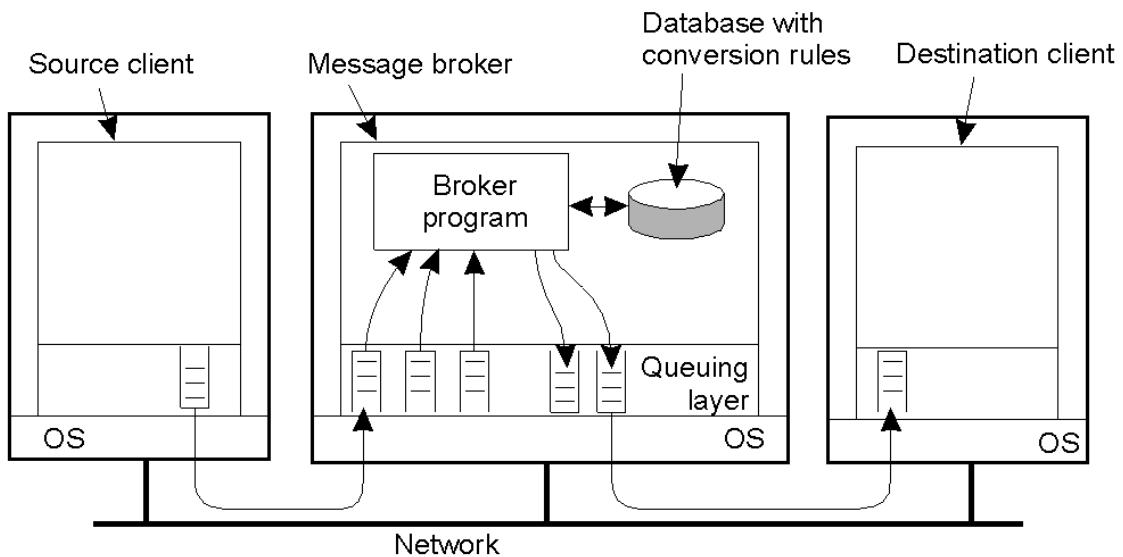
desumere dal nome di B che il messaggio deve essere inoltrato al router **R2**. In questo modo, è necessario aggiornare soltanto i router quando vengono aggiunte o rimosse delle code, mentre tutti gli altri gestori devono solo sapere dove si trova il router più vicino.

Questa architettura viene utilizzata quando le reti crescono di dimensione, in questo caso infatti la sua configurazione manuale diventa pressoché ingestibile. La soluzione sta quindi nell'adottare un sistema di **routing dinamici**.

## BROKER DI MESSAGGI

Un'applicazione importante dei sistemi a code di messaggi è l'integrazione di applicazioni nuove ed esistenti in un unico sistema distribuito coerente. L'integrazione richiede che le applicazioni possano **interpretare i messaggi** che ricevono. In pratica, ciò comporta che i messaggi in uscita dal mittente siano nello **stesso formato** di quelli del destinatario.

Sebbene siano stati definiti alcuni messaggi comuni per specifici domini applicativi, l'approccio generale è di imparare a convivere con i diversi formati e provare a rendere la **conversione** il più semplice possibile. Vengono introdotti quindi i **broker di messaggi**. Un **broker** agisce da **getway** a livello applicativo. Il suo obiettivo principale è quello di convertire i messaggi in ingresso in modo tale che siano comprensibili dell'applicazione destinataria.



Si noti che in un sistema a code il broker non è altro che un'altra applicazione come mostrato in figura, non è quindi considerato come parte integrante di un sistema di code.

Un broker può essere semplice quanto un traduttore di messaggi. Tuttavia più comune è l'utilizzo di un broker per **l'integrazione di applicazioni aziendali**. In questo caso, invece di semplicemente convertire i messaggi un broker deve trovare una corrispondenza tra le applicazioni basandosi sui messaggi che si sono scambiati. In tale modello, chiamato **publish/subscribe**, le applicazioni inviano messaggi sotto forma di **publishing**. In particolare, esse possono **pubblicare** un messaggio sull'argomento X che viene quindi inviato al broker. Le applicazioni che sono interessate ai messaggi sull'argomento X, che hanno cioè **sottoscritto(subscribed)** questi messaggi, li riceveranno dal broker.

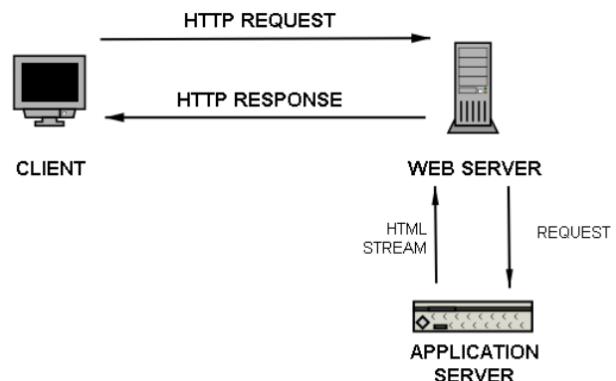
# WEB APPLICATION

## PERCHE' IL WEB

Disponiamo di un ambiente standard, diffuso su larga scala, che utilizza un protocollo conosciuto ed affidabile (TCP/IP); inoltre le applicazioni sviluppate nel web sono indipendenti dalle piattaforme, poiché per utilizzarle basta disporre di un browser, comunicante dunque via http. L'internet ci offre inoltre un'infrastruttura completa, dinamica ed efficiente, sfruttabile con strumenti performanti: evoluzioni di [HTML](#), [CGI](#), [JavaScript](#), [Java](#), [PHP](#), ...

Come accennato la comunicazione avviene tramite http, dunque conversazioni senza stato, dunque ogni richiesta è un messaggio autonomo; è possibile tuttavia creare sessioni di lavoro utilizzando i [cookie](#) e [campi nascosti](#) nei messaggi http.

Le operazioni di input e output avvengono tramite l'uso di [FORM](#) e pagine HTML.



Per realizzare un'applicazione , il [Web Server](#) utilizza un [Application Server](#) caratterizzata da un particolare protocollo di interazione con il Web Server.

## SERVER SIDE

La computazione avviene lato server e può avvenire tramite [programmi compilati](#) o [script interpretati](#); nel caso dei programmi compilati il web server si limita ad invocare, su richiesta del client un eseguibile , il quale può essere scritto in un qualsiasi linguaggio che supporti l'interazione con il web server (Es: Java, C#, C++); nel caso di esecuzione di script, il web server utilizza un motore ([engine](#)) in esso integrato, in grado di interpretare il linguaggio di scripting utilizzato(Es: PHP, Python, Perl);

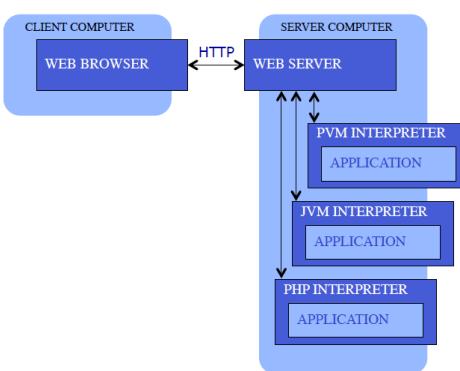
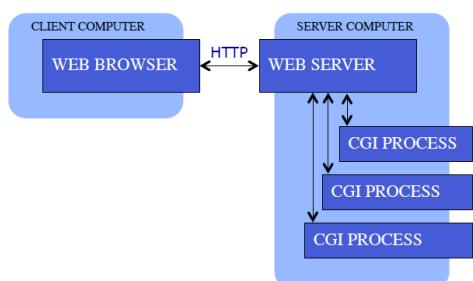
**Osservazione:** utilizzando un linguaggio interpretato (scripting) si perde in velocità di esecuzione, ma si guadagna in facilità di scrittura dei programmi;

## MODELLO CGI

Il [CGI](#) è la prima forma di elaborazione [lato server](#) implementata: quando ad un web server arriva la richiesta di un documento CGI (solitamente con estensione .cgi, .exe o .pl) il server esegue il programma richiesto e al termine invia al web browser l'output del programma. Il file CGI è un

semplice programma già compilato (codice oggetto) e la risposta viene acquisita attraverso standard output. L'acquisizione dei parametri può avvenire attraverso variabili d'ambiente, passaggio di parametri sulla riga di comando o lo standard input a seconda della mole di dati e delle scelte del programmatore. Il processo CGI implementa un interprete per il linguaggio utilizzato.

Esempi: Python, [Java/Servlet](#), PHP.



## Vantaggi:

- Portabilità
- Utilizzo di strutture ben definite
- Serve programmare solo le logiche delle applicazioni

----- o ----- o -----

# JAVA SERVLET

## Client side

Lato client avremo niente di meno che una pagina web HTML, possiamo richiedere e inviare dati tramite form;

## Server Side

Occupiamoci ora della programmazione lato server; le **Java Servlet** sono delle piccole applicazioni Java residenti sul server (che può essere per esempio Apache, o Tomcat), gestite in modo automatico da un **container** o **engine**, dotate di un'interfaccia che definisce il set di metodi (ri)definibili. Il **container** controlla le servlet (attiva/disattiva) in base alle richieste di client.

Utilizzando le Java Servlet disporremmo dunque di semplicità di implementazione e standardizzazione, ma dovremo sottostare ad una rigidità del modello dettate da quest'ultime. Le servlet sono residenti in memoria, mantengono dunque uno stato e permettono la comunicazione con altre servlet.

Utilizzando il protocollo http anche in questo caso non sarà prevista una comunicazione persistente, se non esplicitamente richiesta attraverso i metodi già citati, come i **cookies** che permettono di gestire sessioni, o direttamente **HttpSession** gestito automaticamente dal container.

Ogni servlet implementa l'interfaccia **javax.servlet.Servlet**, con 5 metodi.

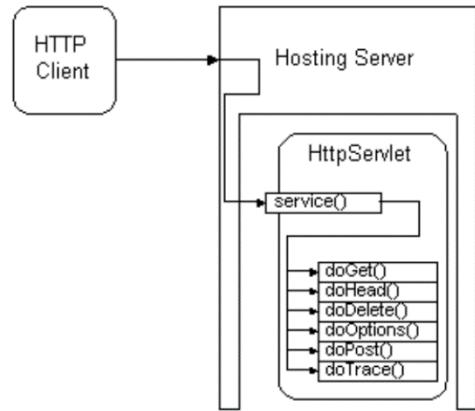
- void init(ServletConfig config)
  - Inizializza la servlet
- ServletConfig getServletConfig()
  - Restituisce i parametri di inizializzazione e il ServletContext che dà accesso all'ambiente
- void service(ServletRequest request, ServletResponse response)
  - Invocato per gestire le richieste dei client
- String getServletInfo()
  - Restituisce informazioni tipo autore e versione
- void destroy()
  - Chiamata quando la servlet termina (es: per chiudere un file o una connessione con un database)

L'interfaccia è solo la dichiarazione dei metodi che, per essere utilizzabili, devono essere implementati in una classe. Questo semplifica la scrittura delle servlet vere e proprie in quanto basta implementare (ridfinendoli) solo i metodi che ci interessano.

## Classe astratta HttpServlet

Implementa `service` in modo da invocare i metodi per servire le richieste del web; metodo `doX`, dove X è un metodo http.

Anche i parametri sono stati adattati al protocollo HTTP, cioè consentono di ricevere (inviare) messaggi http leggendo (scrivendo) i dati nell'head e nel body di un messaggio.



I metodi principali sono:

- `String getParameter(String name)`
  - Restituisce il valore dell'argomento name
- `Enumeration getParameterNames()`
  - Restituisce l'elenco dei nomi degli argomenti
- `String[] getParametersValues(String name)`
  - Restituisce i valori dell'argomento name
- `Cookie[] getCookies()`
  - Restituisce i cookies del server sul client
- `void addCookie(Cookie cookie)`
  - Aggiunge un cookie nell'intestazione della risposta
- `HttpServletRequest getSession(boolean create)`
  - Una HttpServletRequest identifica il client.
  - Viene creata se create=true
- `void setContentType(String type)`
  - Specifica il tipo MIME della risposta per dire al browser come visualizzare la risposta
    - Es: "text/html" dice che e' html
- `ServletOutputStream getOutputStream()`
  - Restituisce lo stream di byte per scrivere la risposta
- `PrintWriter getWriter()`
  - Restituisce lo stream di caratteri per scrivere la risposta

## ESECUZIONE DI UNA SERVLET

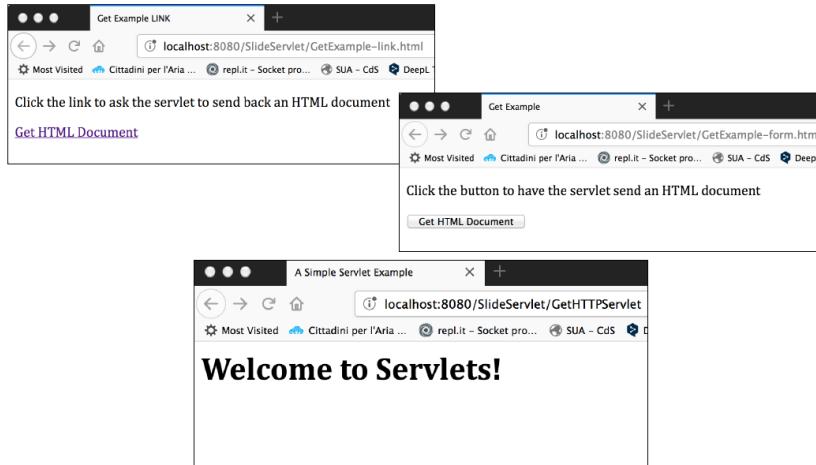
Al lancio di una java servlet viene creata un'istanza di quest'ultima, condivisa da tutti i client. Ogni richiesta genera un Thread che esegue la `doXXX` appropriata.

## CICLO DI VITA DI UNA SERVLET

Una servlet viene **creata** dal container quando viene effettuata la prima chiamata, viene quindi invocato il metodo `init()` per inizializzazioni specifiche. Una servlet viene **distrutta** quando non ci sono servizi in esecuzione o quando è scaduto il timeout predefinito. Viene dunque invocato il metodo `destroy()` per terminare correttamente la servlet.

E' importante che il Container e le richieste dei client si sincronizzino sulla **terminazione**, infatti alla scadenza del timeout potrebbe essere ancora in esecuzione la `service()`; bisogna dunque tener traccia dei Thread in esecuzione, progettare il metodo `destroy()` in modo da notificare lo shutdown e attendere il completamento del metodo `service()` progettare i metodi lunghi in modo che verifichino periodicamente se è in corso uno shutdown e comportarsi di conseguenza.

## Esempio: GET



```

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <meta charset="UTF-8">
5. <title>Get Example</title>
6. </head>
7. <body>

8. <form action="http://localhost:8080/SlideServlet/GetHTTPServlet"
9.      method="GET"> Metodo di invio
10.     <p>Click the link to ask the servlet to send back an HTML document</p>
11.     <input type="submit" value="Get HTML Document">
12.   </form>
L'applicazione che processa il form
13. </body>
14. </html>
```

Client side

Crea il bottone di invio

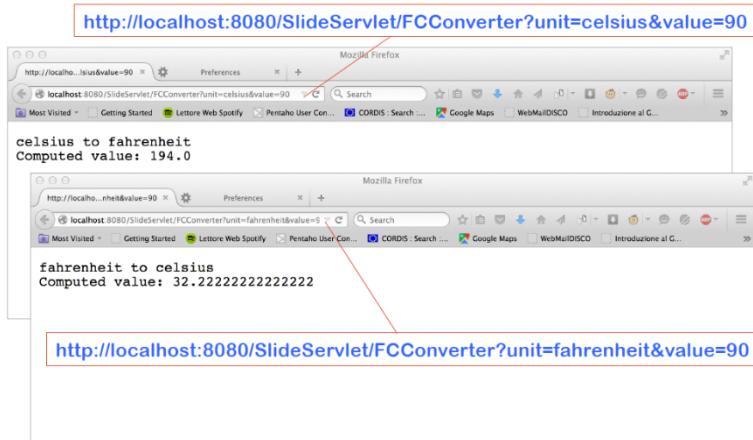
Etichetta esposta

Server Side

```

1. // da Internet e WWW - How to program, Dietel&Dietel, Prentice Hall
2. // Creating and sending a page to the client
3. public class GetHTTPServlet extends HttpServlet {
4.     public void doGet( HttpServletRequest request,
5.                        HttpServletResponse response )
6.             throws ServletException, IOException {
7.         PrintWriter output;
8.         response.setContentType( "text/html" ); // content type
9.         output = response.getWriter(); // get writer
10.        // create and send HTML page to client
11.        StringBuffer buf = new StringBuffer();
12.        buf.append( "<html><head><title>\n" );
13.        buf.append( "A Simple Servlet Example\n" );
14.        buf.append( "</title></head><body>\n" );
15.        buf.append( "<H1>Welcome to Servlets!</H1>\n" );
16.        buf.append( "</body></html>" );
17.        output.println( buf.toString() );
18.        output.close(); // close PrintWriter stream
    }
```

## Esempio: GET con parametri



## Server Side

```
1. public class FCConverter extends HttpServlet {
2.     private String convertCtoF(Double celsius) {
3.         Double fahrenheit;
4.         fahrenheit = ((celsius * 9) / 5) + 32;
5.         return Double.toString(fahrenheit);
6.     }
7.     private String convertFtoC(Double fahrenheit) {
8.         Double celsius;
9.         celsius = (fahrenheit - 32)*5/9;
10.        return Double.toString(celsius);
11.    }
12.    protected void doGet(HttpServletRequest request, HttpServletResponse response)
13.            throws ServletException, IOException {
14.        String result;
15.        String unit=request.getParameter("unit");
16.        Double value=Double.parseDouble(request.getParameter("value"));
17.
18.        if( unit.equals("fahrenheit")) {
19.            response.getWriter().append("Fahrenheit to celsius \n");
20.            result=convertFtoC(value);
21.        } else {
22.            response.getWriter().append("celsius to fahrenheit \n");
23.            result=convertCtoF(value);
24.        }
25.
26.        response.getWriter().append("Computed value: ").append(result);
27.    }
28. }
```

## JSP (JAVA SERVER PAGES)

Si tratta di una tecnologia per la creazione delle applicazioni web. Essa specifica l'interazione tra un contenitore / server ed un insieme di "pagine" che presentano informazioni all'utente. Le pagine sono costituite da tag tradizionali (HTML, XML, WML, ...) e da tag applicativi che controllano la generazione del contenuto. Rispetto ai **servlet** facilitano la separazione tra **logica applicativa** e **presentazione**, separando la parte dinamica delle pagine dal template HTML statico, infatti il codice JSP è incluso in opportuni tag ( “<%” e “%>”).

### Esempio:

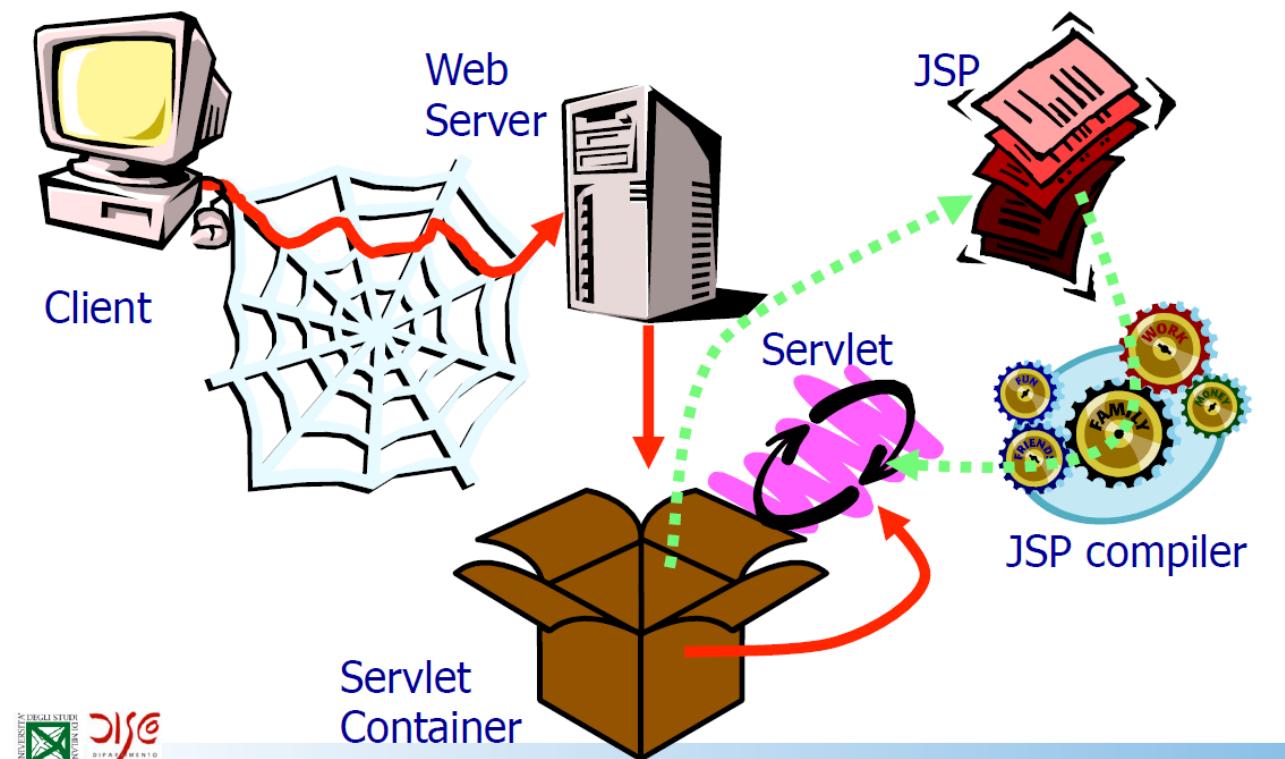
Una pagina che visualizza “Grazie per la scelta di Internet Guida Pratica” quando l'utente si connette all'URL

<http://host/OrderConfirmation.jsp?title=Internet+Guida+Pratica>

contiene

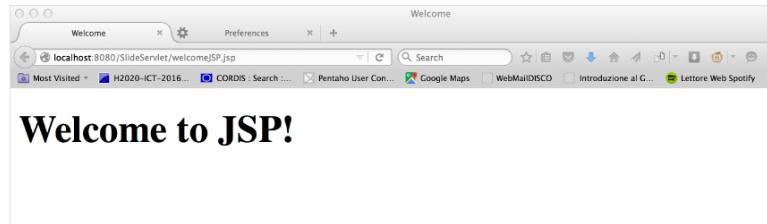
*Grazie per la scelta di <i><%= request.getParameter("title") %> </i>*

La pagina viene convertita automaticamente in una servlet java la prima volta che viene richiesta.



## Esempio:

```
1. <%@ page language="java" contentType="text/html; charset=UTF-8"
2.     pageEncoding="UTF-8"%>
3. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://
www.w3.org/TR/html4/loose.dtd">
4. <html>
5. <head>
6. <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7. <title>Welcome</title>
8. </head>
9. <body>
10.    <h1><%= "Welcome to JSP!" %><h1>
11. </body>
12. </html>
```



## ELEMENTI DI UNA JSP

- **Tamplate Text:** parti statiche della pagine , elementi html
- **Commenti:** <%-- questo è un commento --%>
- **Direttive:** <%@ direttiva ... di compilazione %>
  - **Page:** liste di attributi/valore , valgono per la pagina in cui sono inseriti;  
Esempio:  

```
<%@ page import="java.util.*" buffer="16k" %>
<%@ page import="java.math.*, java.util.*" %>
<%@ page session="false" %>
```
  - **Include:** include in compilazione pagine HTML o JSP  
Esempio:  

```
<%@ include file="copyright.html" %>
```
  - **Taglib:** dichiara tag definiti dall'utente implementando opportune classi  
Esempio:  

```
<%@ taglib uri="TableTagLibrary" prefix="table"%>
<table:loop> ... </table:loop>
```
  - **Forward:** determina l'invio della richiesta corrente, eventualmente aggiornata con ulteriori parametri, all'URL indicata  
Esempio:  

```
<jsp:forward page="login.jsp" %>
<jsp:param name="username" value="user" />
<jsp:param name="password" value="pass" />
</jsp:forward>
```
  - **UseBean:** localizza ed istanzia (se necessario) una javaBean nel contesto specificato; il contesto può essere la pagina, la richiesta, la sessione, l'applicazione ...  
Esempio:  

```
<jsp:useBean id="cart" scope="session" class="ShoppingCart" />
```
- **Azioni:** descritte in linguaggio XML: <tag attributi> body </tag>

- **Elementi di scripting:** sono istruzioni nel linguaggio specificato nelle direttive, possono essere di 3 tipi: **scriptlet, declaration, expression**
  - **Declaration:** `<%! declaration [declaration] ...%>`, variabili o metodi usati nella pagina  
Esempio:  
`<%! int[] v= new int[10]; %>`
  - **Expression:** `<%= expression %>`, un'espressione nel linguaggio di scripting che viene valutata e sostituita con il risultato  
Esempio:  
`<p>La radice di 2 vale <%= Math.sqrt(2.0) %></p>`
  - **Scriptlet:** `<% codice %>`, frammenti di codice che controllano la generazione della pagina, valutati alla richiesta  
Esempio:  
`<table>  
 <% for (int i=0; i < v.length; i++) { %>  
 <tr><td> <%= v[i] %></td></tr>  
 <% } %>  
</table>`

Le variabili valgono per la **singola esecuzione**, ciò che viene scritto sullo stream di output sostituisce lo scriptlet

### Esempio:

```

1. 1. <!DOCTYPE html>
2. 2. <html>
3. 3. <head>
4. 4. <meta charset="UTF-8">
5. 5. <title>JSP Example</title>
6. 6. </head>
7. 7. <body>

8. 8. <form action="http://localhost:8080/SlideServlet/FC.jsp"
9. 9.     method="GET">
10. 10.    Converter <BR><BR>
11. 11.    <input type="radio" name="unit" value="celsius">Celsius<BR>
12. 12.    <input type="radio" name="unit" value="fahrenheit">Fahrenheit<BR>
13. 13.    value <input type="text" name="value" ><BR><BR>
14. 14.    <input type="submit" value="send">
15. 15.    <input type="reset">
16. 16. </form>

17. 17. </body>
18. 18. </html>

```

```

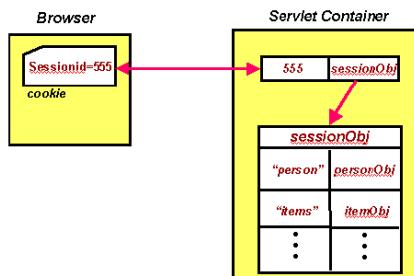
1. <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
3. <html><head><meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
4. <title>F2C converter</title></head>
5. <body>
6.   <%@ page import = "java.text.DecimalFormat" %>
7.   <% Double result;
8.   String unit = request.getParameter("unit");
9.   Double value=Double.parseDouble(request.getParameter("value")); %>
10.  <h1> <% if( unit.equals("fahrenheit")) { %>
11.    <%= "Fahrenheit to Celsius" %>
12.    <% result=(value - 32)*5/9;
13.  } else { %>
14.    <%= "Celsius to Fahrenheit" %>
15.    <% result=((value * 9) / 5) + 32; %>
16.  </h1>
17.  <% DecimalFormat twoDigits = new DecimalFormat( "#0.00" ); %>
18.  <h2> <%= unit %> : <%= twoDigits.format(value) %> </h2>
19.  <h2> <% if( unit.equals("fahrenheit")) { %>
20.    <%= "Fahrenheit: " %>
21.    <% result=(value - 32)*5/9;
22.  } else { %>
23.    <%= "Celsius: " %>
24.    <% result=((value * 9) / 5) + 32; %>
25.    <%= twoDigits.format( result ) %>
26.  </h2>
27. </body>
28. </html>

```

5

Il linguaggio di script ha lo scopo di interagire con **oggetti** Java e gestire le eccezioni Java. Gli **oggetti** possono essere creati implicitamente usando le direttive JSP, esplicitamente con le azioni, o direttamente usando uno script (caso raro). Gli oggetti hanno un attributo che ne definisce lo **scope**

### Oggetto HttpSession



### Esempi

- Memorizzazione
 

```
<% Foo foo = new Foo(); session.putValue("foo",foo); %>
```
- Recupero
 

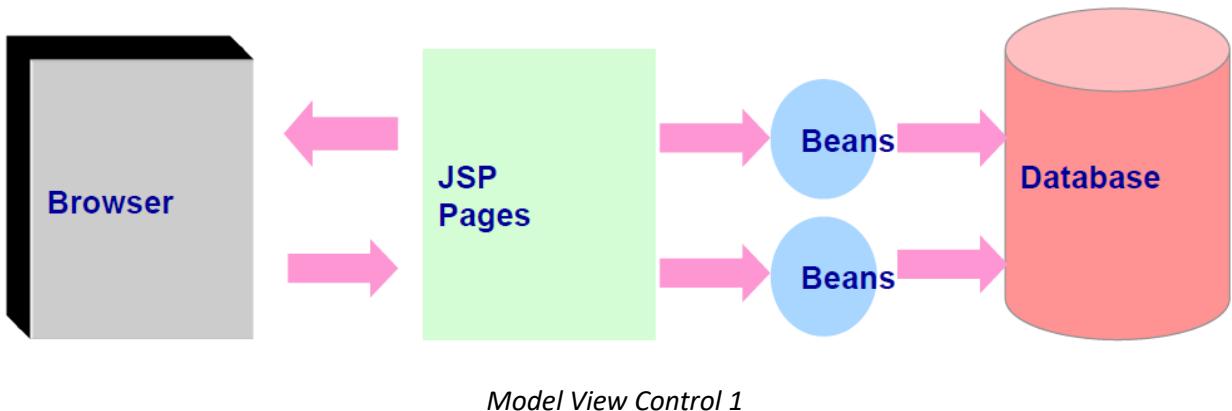
```
<% Foo myFoo = (Foo) session.getValue("foo"); %>
```
- Esclusione di una pagina dalla sessione
 

```
<%@ page session="false" %>
```

----- 0 ----- 0 -----

## IL PATTERN MVC (Model View Control)

L' **MVC** si preclude come scopo quello di imporre una separazione tra dati, controllo e visualizzazione. Come funziona: il client richiede via http un file .JSP, quest'ultimo viene interpretato e accede a componenti lato-server (**Java Beans**, **Servlet**) che generano contenuti dinamici. Il risultato viene spedito al client in forma di pagine HTML. Un'architettura tipica è riportata nella figura sottostante



## JAVA BEAN

Le **JavaBeans** sono **classi** scritte in linguaggio di programmazione Java secondo una particolare convenzione. Sono utilizzate per incapsulare più oggetti in un oggetto singolo (il bean), cosicché tali oggetti possano essere passati come un singolo oggetto bean invece che come multipli oggetti individuali. Specifiche per la classe:

- Il costruttore deve essere privo di parametri
- I campi dovrebbero essere private
- Le sue proprietà devono essere accessibili usando get, set, is, seguendo una convenzione standard per i nomi, in particolare `setXxxx` e `getXxxx/isXxxx` dove xxx rappresenta una **proprietà**

### Esempio:

```
class Book{  
    private String title;  
    private boolean available;  
    void setTitle(String t) ...;  
    String getTitle() ...;  
    void setAvailable(boolean b) ...;  
    boolean isAvailable () ...;  
}
```

## JSP E JAVABEAN

### Accedere ad un Bean (inizializzazione):

```
<jsp:useBean id="user" class="com.jguru.Person" scope="session" />
<jsp:useBean id="user" class="com.jguru.Person" scope="session">
    <% user.setDate(DateFormat.getDateInstance().format(new Date())); //etc.. %>
</jsp:useBean>
```

Esempio:

```
<jsp:useBean id="Attore" class="MyThread" scope="session" type="Thread"/>
```

Lo **scope** determina la **vita del bean**:

- ✓ **Page**: è lo scope di default, viene messo in pageContext ed acceduto con getAttribute
- ✓ **Request**: viene messo in ServletRequest ed acceduto con getAttribute
- ✓ **Session e application**: se non esiste un bean con lo stesso id, ne viene creato uno nuovo

Il **type** permette di assegnargli una superclasse. Al posto della classe è possibile utilizzare il nome del bean

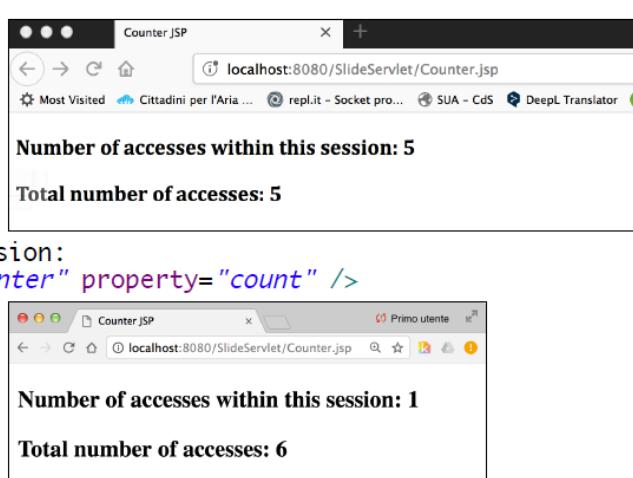
### Accedere alle proprietà

```
<jsp:getProperty name="user" property="name" />
<jsp:setProperty name="user" property="name" value="jGuru" />
<jsp:setProperty name="user" property="name" value="<%="expression %>" />
```

Esempio:

Counter.jsp

```
1.  <%@ page import="ititis.mvc.CounterBean"%>
2.  <jsp:useBean id="session_counter" class="ititis.mvc.CounterBean"
3.      scope="session" />
4.  <jsp:useBean id="app_counter" class="ititis.mvc.CounterBean"
5.      scope="application" />
6.  <%
7.      session_counter.increaseCount();
8.      synchronized (page) {
9.          app_counter.increaseCount();
10.     }
11.  %>
12.  <h3>
13.      Number of accesses within this session:
14.      <jsp:getProperty name="session_counter" property="count" />
15.  </h3>
16.  <p></p>
17.  <h3>
18.      Total number of accesses:
19.      <%
20.      synchronized (page) {
21.          %
22.          <jsp:getProperty name="app_counter" property="count" />
23.          %
24.          }
25.          %
26.      </h3>
```

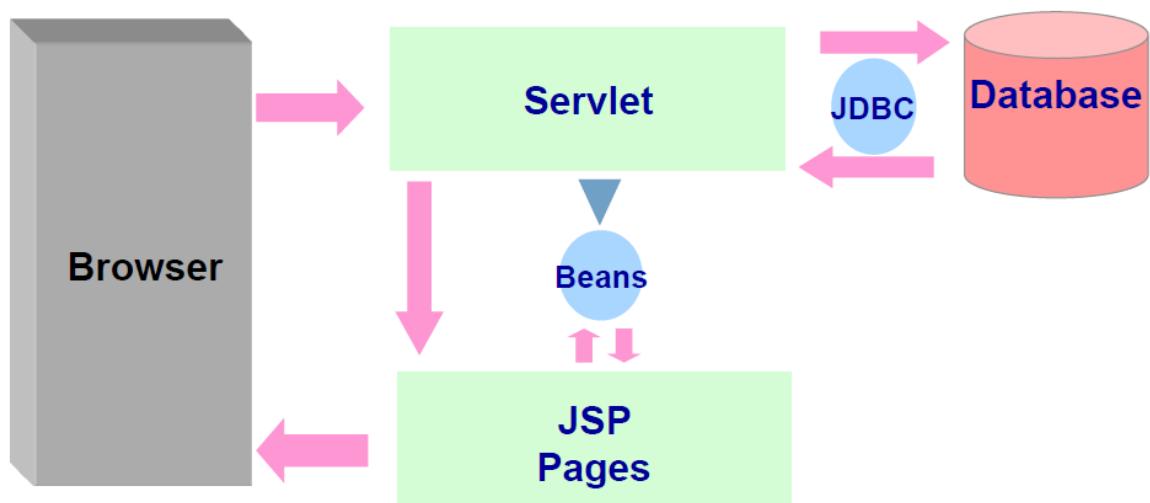


## CounterBean.java

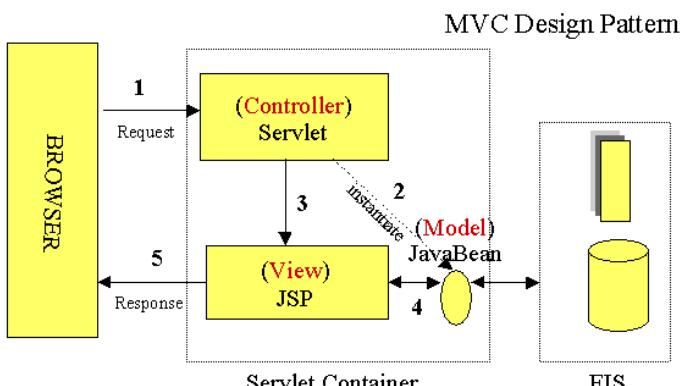
```
1. public class CounterBean {  
2.     //declare a integer for the counter  
3.     private int count;  
  
4.     public int getCount() {  
5.         //return count  
6.         return count;  
7.     }  
  
8.     public void increaseCount() {  
9.         //increment count  
10.        count++;  
11.    }  
12.}
```

## MODEL VIEW CONTROL 2

Come funziona: la richiesta viene inviata da una Java Servlet che genera i dati dinamici richiesti dall'utente, li mette a disposizione della pagina jsp come Java "Beans"; la servlet chiama una pagina .jsp, che legge i dati dei beans e organizza la presentazione in HTML che invierà all'utente.



## Architettura MVC2



**Model:** oggetto dell'applicazione;

**View:** è la sua presentazione sullo schermo;

**Controller:** definisce il modo in cui l'interfaccia utente reagisce all'input dell'utente

# WEB SERVICE

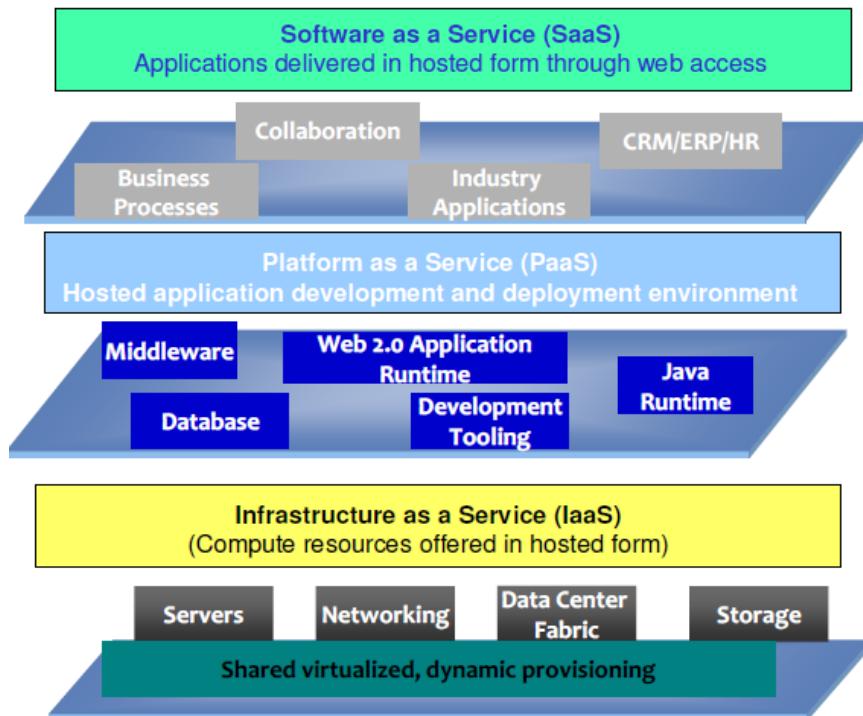
## PANORAMICA

I **web services** sono applicazioni modulari pubblicate, allocate e invocate attraverso il web. Le applicazioni sono incapsulate, debolmente accoppiate e legate dinamicamente tra loro. I servizi sono **componenti indipendenti**, con un'interfaccia nota, un unico **access point (URL)** e lo scambio di dati basato su **documenti**. Il **linguaggio di descrizione** è standard e il middleware può gestire i servizi. I parametri vengono scambiati usando rappresentazioni standard come XML.

## ARCHITETTURA BASE

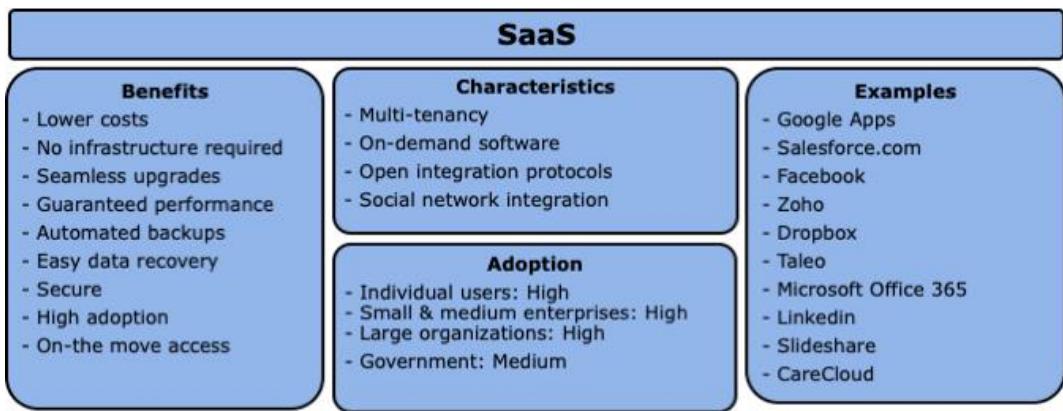
L'architettura di base prende il nome di **SOA (Software Oriented Architecture)**, un'architettura software adatta a supportare l'uso di servizi Web per garantire l'interoperabilità tra diversi sistemi così da consentire l'utilizzo delle singole applicazioni come componenti del processo di business e soddisfare le richieste degli utenti in modo integrato e trasparente. E' composta da una discovery agency che indicizza le descrizioni e fornisce l'elenco dei servizi, la richiesta del servizio e un provider. Le operazioni sono **pubblicazione, ricerca e interazione**.

## XaaS STACK



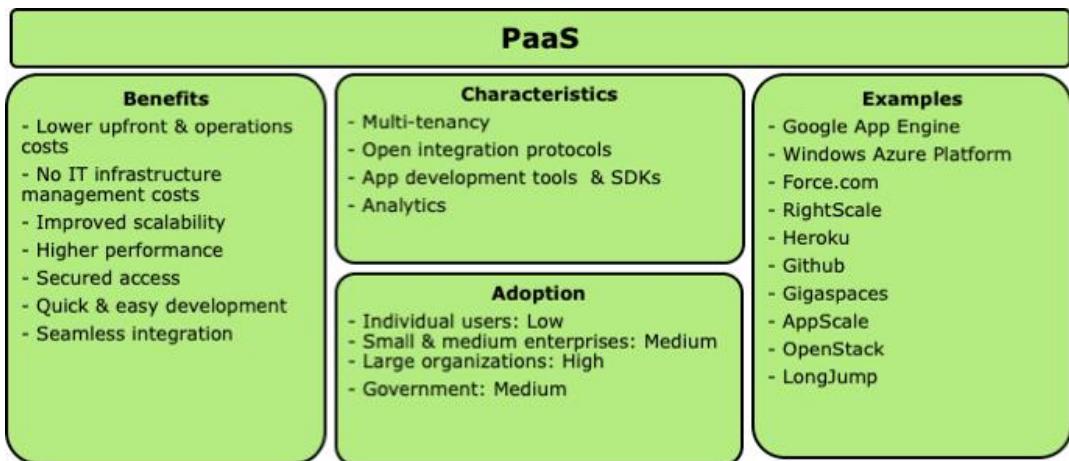
**XaaS** ha tre diverse architetture a strati, di cui ognuna virtualizza strati diversi, in modo che le applicazioni siano create su una VM (che si appoggia sul sistema operativo) e poi eseguite. I tre livelli classici di un'architettura sono. In ordine: **SaaS, PaaS, IaaS**.

## SaaS (Software as a Service)



Fornisce agli utenti un'applicazione software completa o l'interfaccia utente di quest'ultima. Il fornitore di servizi cloud gestisce l'infrastruttura cloud sottostante inclusi server, rete, sistemi operativi, storage e applicazioni, e l'utente non è a conoscenza dell'architettura sottostante del cloud. Le applicazioni vengono fornite all'utente tramite un'interfaccia **thin client** (ad esempio un browser), e sono **indipendenti** dalla piattaforma e accessibili da vari dispositivi client come workstation, laptop, tablet e smartphone, con diversi sistemi operativi.

## PaaS (Platform as a Service)



Offre agli utenti la possibilità di **sviluppare e distribuire app nel cloud** utilizzando gli strumenti di sviluppo, API, librerie software e servizi forniti dal cloud. Gli utenti stessi sono responsabili dello sviluppo, dell'implementazione configurazione e gestione di applicazioni sull'infrastruttura cloud.

## IaaS (Infrastructure as a Service)

IaaS		
<b>Benefits</b> <ul style="list-style-type: none"><li>- Shift focus from IT management to core activities</li><li>- No IT infrastructure management costs</li><li>- Pay-per-use/pay-per-go pricing</li><li>- Guaranteed performance</li><li>- Dynamic scaling</li><li>- Secure access</li><li>- Enterprise grade infrastructure</li><li>- Green IT adoption</li></ul>	<b>Characteristics</b> <ul style="list-style-type: none"><li>- Multi-tenancy</li><li>- Virtualized hardware</li><li>- Management &amp; monitoring tools</li><li>- Disaster recovery</li></ul>	<b>Examples</b> <ul style="list-style-type: none"><li>- Amazon Elastic Compute Cloud (EC2)</li><li>- RackSpace</li><li>- GoGrid</li><li>- Eucalyptus</li><li>- Joyent</li><li>- Terremark</li><li>- OpSource</li><li>- Savvis</li><li>- Nimbus</li><li>- Enamoly</li></ul>
	<b>Adoption</b> <ul style="list-style-type: none"><li>- Individual users: Low</li><li>- Small &amp; medium enterprises: Medium</li><li>- Large organizations: High</li><li>- Government: High</li></ul>	

Fornisce agli utenti la capacità di **fornire risorse di elaborazione e archiviazione**. Queste risorse vengono fornite dagli utenti come **istanze della macchina virtuale**. Gli utenti possono avviare, arrestare, configurare e gestire le istanze della macchina virtuale. Le risorse virtuali fornite dagli utenti sono fatturate in base a paradigmi pay-per-use / pay-as-you-go.

**Esempi:** si desidera eseguire un processo batch ma non si dispone dell'infrastruttura necessaria per eseguirlo in modo tempestivo; oppure: si desidera ospitare un sito web ma solo per pochi giorni.

## PROCESSI WEB

I **processi web** sono generati dalla composizione di web services conosciuti. Questi servizi provengono da diversi provider, e sono assemblati in sequenza.

**Orchestrazione:** descrive come i web services possono interagire tra di loro tramite messaggi, includendo la logica di business e l'ordine di esecuzione delle interazioni dalla prospettiva di un singolo endpoint coordinatore.

**Coreografia:** descrive la sequenza di messaggi che coinvolgono più parti e risorse dalla prospettiva di ognuna delle parti. Definisce lo stato condiviso delle interazioni tra entità di business.

—o—o—o—

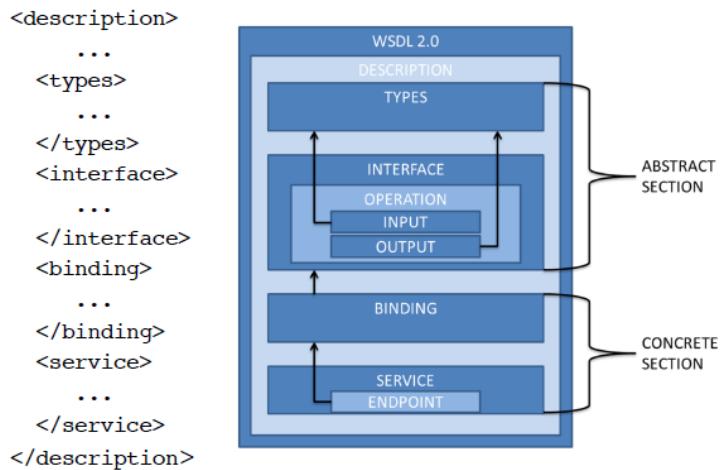
## WSDL (Web Service Description Language)

È un linguaggio basato su XML per la creazione di “documenti” per la descrizione dei web services (“cosa, come, dove”). In particolare descrive quattro importanti informazioni:

- Informazioni sull'**interfaccia** che descrivono tutte le operazioni disponibili pubblicamente;
- Dichiarazioni sul **tipo di dati** per tutte le richieste e le risposte dei messaggi;
- Informazioni vincolanti sul **protocollo di trasporto**;
- Informazioni sull'**indirizzo** per localizzare il servizio;

In una descrizione WSDL viene adottato uno schema concettuale che separa con due sezioni una parte di descrizione **“astratta”** in cui vengono descritti i modelli per lo scambio di messaggi, e una **“concreta”** in cui viene eseguito il binding con gli indirizzi di tutti gli endpoint e viene implementata un’interfaccia comune.

## STRUTTURA DI UNA DESCRIZIONE WSDL



- **description:** la **root** di qualsiasi file WSDL 2.0
- **types:** la specifica dei **dati** che devono essere scambiati tra il fornitore del servizio e il consumatore
- **interface:** descrizione delle **operazioni disponibili** nel WS e quali **messaggi** dovrebbero essere scambiati tra il fornitore e il consumatore del servizio per ogni operazione (request/response); questo elemento è usato anche per descrivere qualsiasi messaggio di errore che potrebbe verificarsi
- **binding:** descrizione di **come** è possibile accedere al servizio sulla rete; quest'ultimo solitamente lega il WS al protocollo conosciuto http
- **service:** descrizione di **dove** è possibile accedere al servizio attraverso la rete, solitamente contenente un URL e porta di quest'ultimo

## <description>

XML namespace per WSDL 2.0;

Namespace per il servizio;

Prefisso per targetNameSpace, questo attributo deve essere impostato come targetNameSpace utilizzando lo stesso URI;

Schema Target Namespace Uri, utilizzato in <types> per le dichiarazioni dei tipi del nostro web service;

WSDL SOAP URI, utilizzato nel <binding>

SOAP URI della versione SOAP utilizzata

```

<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns = "http://www.w3.org/ns/wsdl"
    targetNamespace = "http://yoursite.com/MyService"
    xmlns:tns = "http://yoursite.com/MyService"
    xmlns:stns = "http://yoursite.com/MyService/schema"
    xmlns:wsoap = "http://www.w3.org/ns/wsdl/soap"
    xmlns:soap = "http://www.w3.org/2003/05/soap-envelope"
    >
    ...
</description>

```

## <types>

Describe i diversi tipi di dati che saranno utilizzati dal WS; un WS solitamente ha uno o più tipi in input e in output, e uno o più tipi di errore. Se il WS ha molte funzioni (operazioni) dichiarate, ognuna di queste potrebbe avere il proprio set di tipi in input e output e di errore. I tipi di dati possono essere dichiarati in qualsiasi linguaggio purché sia supportato dal WS, ma solitamente sono specificati nello schema XML.

```
<types>
  <xss:schema
    xmlns:xs = "http://www.w3.org/2001/XMLSchema"
    targetNamespace = "http://yoursite.com/MyService/schema"
    xmlns = "http://yoursite.com/MyService/schema" >
    <xss:element name="checkServiceStatus" type="tCheckServiceStatus" />
    <xss:complexType name="tCheckServiceStatus" >
      <xss:sequence>
        <xss:element name = "checkDate" type = "xs:date" />
        <xss:element name = "serviceName" type = "xs:string" />
      </xss:sequence>
    </xss:complexType>
    <xss:element name = "checkServiceStatusResponse" type = "xs:double" />
    <xss:element name = "dataError" type = "xs:string" />
  </xss:schema>
</types>
```

## <binding>, <operation>, <fault>

```
<binding name = "myServiceInterfaceSOAPBinding"
  interface = "tns:myServiceInterface"
  type = "http://www.w3.org/ns/wsdl/soap"
  wsoap:protocol = "http://www.w3.org/2003/05/soap/bindings/HTTP/">
  <operation ref = "tns:checkServiceStatusOp"
    wsoap:mep = "http://www.w3.org/2003/05/soap/mep/soap-response" />
    <fault ref = "tns:dataFault"
      wsoap:code = "soap:Sender"/>
  </binding>
```

L'attributo **name** definisce il nome del binding; con quest'ultimo è possibile fare riferimento ad esso quando si definisce un endpoint del servizio. Ogni nome deve essere univoco nel target namespace di WSDL 2.0.

L'attributo **interface** indica il nome di un'interfaccia già definita.

L'attributo **type** definisce quale formato dei messaggi verrà utilizzato. In questo esempio è SOAP.

L'attributo **wsoap:protocol** definisce il protocollo di trasporto sottostante. In questo esempio i messaggi verranno trasportati utilizzando HTTP.

L'attributo **ref** dell'elemento **operation** fa riferimento ad un'operazione specifica (già definita nella sezione interfaccia).

L'attributo **wsoap:mep** definisce il modello di scambio dei messaggi per SOAP

L'attributo **ref** di **fault** definisce a quale errore (già definito nella sezione **interface**) si riferirà

L'attributo **wsoap:code** dell'elemento **fault** definisce il codice errore che attiverà l'invio di questo messaggio di errore

### <service>, <endpoint>

```
<service name = "myService"  
        interface = "tns: myServiceInterface">  
    <endpoint name = "myServiceEndpoint"  
              binding = "tns:myServiceInterfaceSOAPBinding"  
              address = "http://yoursite.com/MyService" />  
</service>
```

L'attributo **binding** specifica quale binding precedentemente definito verrà usato da questo endpoint

L'attributo **address** specifica l'indirizzo fisico in cui il servizio sarà disponibile

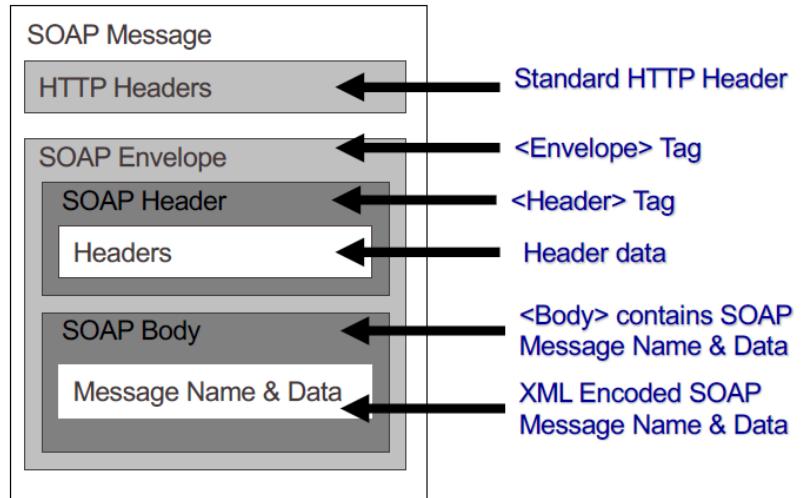
-----o-----o-----

## SOAP

Nell'architettura **SOA** la comunicazione avviene tramite uno dei protocolli internet, e lo scambio di messaggi utilizza estensioni **SOAP (Simple Object Access Protocol)** che definiscono un modo uniforme per la trasmissione XML-encoded data.

**SOAP** è un protocollo basato su XML che consente a componenti e applicazioni software di comunicare utilizzando i protocolli internet standard (ad esempio http). Il protocollo **SOAP è stateless**. I messaggi SOAP sono incapsulati in frame http; un messaggio SOAP è composto da:

- **SOAP envelope**, definisce il contenuto del messaggio;
- **SOAP header (facoltativo)**, contiene informazioni aggiuntive;
- **Body**, contiene il messaggio vero e proprio;



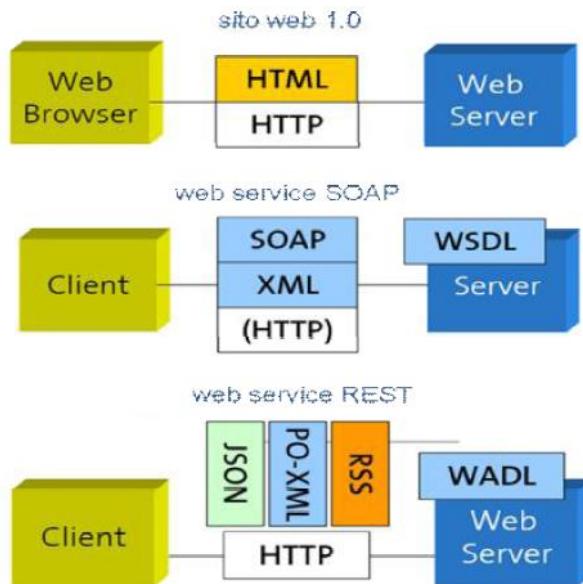
## VANTAGGI

Possibilità di modificare in modo semplice la modalità di interazione tra i servizi, la combinazione in cui essi vengono utilizzati nel processo e rende più agevole l'aggiunta e la modifica di servizi e processi (per rispondere alle esigenze di business).

## RESTful Web Services

Sono particolari WS con architetture che **implementano http in modo nativo**, JSON, PO-XML e RSS nella comunicazione tra un web server e un client; il server pubblica i servizi, utilizzando **WADL (Web Application Description Language)**. L'uso di URI e http rendono REST adatto per l'architettura del web.

### REST vs SOAP



	SOAP	REST
Protocollo di trasporto	Diversi formati (http, TCP, SMTP)	HTTP
Formato dei messaggi	XML-SOAP message format (Envelope, Header, Body)	Diversi formati (XML-SOAP, RSS, JSON)
Identifieri dei servizi	URI e WS-addressing	URI
Descrizione dei servizi	WSDL	Documentazione testuale o WADL
Composizione di servizi	BPEL	Mashup
Service discovery	UDDI	Non supporta standard

**WADL, UDDI e BPEL** sono linguaggi nati per specificare cosa un determinato web service offre. **UDDI (Universal Description Discovery and Integration)** permette di scoprire nuovi servizi e registrare quelli che il proprio web service offre; **BPEL (Business Process Execution Language)** compone servizi in modo da soddisfare i relativi processi di business.

### WEB API

La maggior parte dei servizi esistenti forniscono **API (Application Programming Interface)**, ovvero un insieme di procedure, **per il web**: le chiamate di procedura vengono gestite con http, perché la semantica delle operazioni e dei dati è conosciuta.

## REST

È un insieme di linee guida e best practices che rende centrale il concetto di risorsa. REST è uno **stile architettonico** per sistemi distribuiti, le interfacce trasmettono dati su http senza un livello opzionale come SOAP o la gestione della sessione tramite cookies.

I servizi REST usano HTTP in modo nativo (il che li rende più conformi al modello web rispetto a SOAP), le risorse sono definite da **URI** e manipolate attraverso le loro **rappresentazioni**, possono esserci **rappresentazioni multiple** per la stessa risorsa. I **messaggi** sono **autodescrittivi** e **stateless**: lo stato delle applicazioni è gestito da **manipolazioni di risorse**. Il comportamento delle applicazioni è controllato tramite **hypermedia**.

### Proprietà:

- Il **caching** che riduce il carico del server e il tempo di risposta;
- Il **bilanciamento del carico** tra i server grazie alla comunicazione **stateless**;
- Minore necessità di software specializzati per la **semplicità delle tecnologie**;
- **Meccanismi standard** per l'identificazione, senza bisogno di nomi aggiuntivi;

Il sistema ha anche dei vincoli da rispettare, per risolvere le falte più comuni e avere maggiore fiducia nella costruzione di un software affidabile, scalabile e mantenibile.

## CONTROLLO HYPERMEDIA

Il **controllo hypermedia** è il motore dello stato dell'applicazione sul client (i server mantengono lo stato della risorsa). Un link è una transizione di stato fornita dal server e assunta dal client, contenuta nell'ipertesto. Gli hypermedia provvedono l'ordinamento delle interazioni e l'indipendenza dalle locazioni fisiche, e il controllo hypermedia permette i cambiamenti di stato delle applicazioni governate da client tramite link.

Il web è un grande sistema distribuito **basato su hypermedia**: le principali componenti sono **URI, HTTP e HTML**. Questi principi rendono il web aperto, scalabile, estensibile e facile da comprendere, con l'introduzione di nuove tecnologie per l'evoluzione in modo semplice.

## PROGETTARE APPLICAZIONI RESTful

Per progettare applicazioni RESTful ci sono le seguenti regole:

1. Identificare le risorse (URI);
2. Scegliere le rappresentazioni da utilizzare;
3. Definire la semantica dei metodi;
4. Scegliere i codici di risposta;

Il JDK fornisce gli strumenti base ma ha delle limitazioni nell'uso dei proxy, quindi sono state introdotte librerie per migliorare il package.

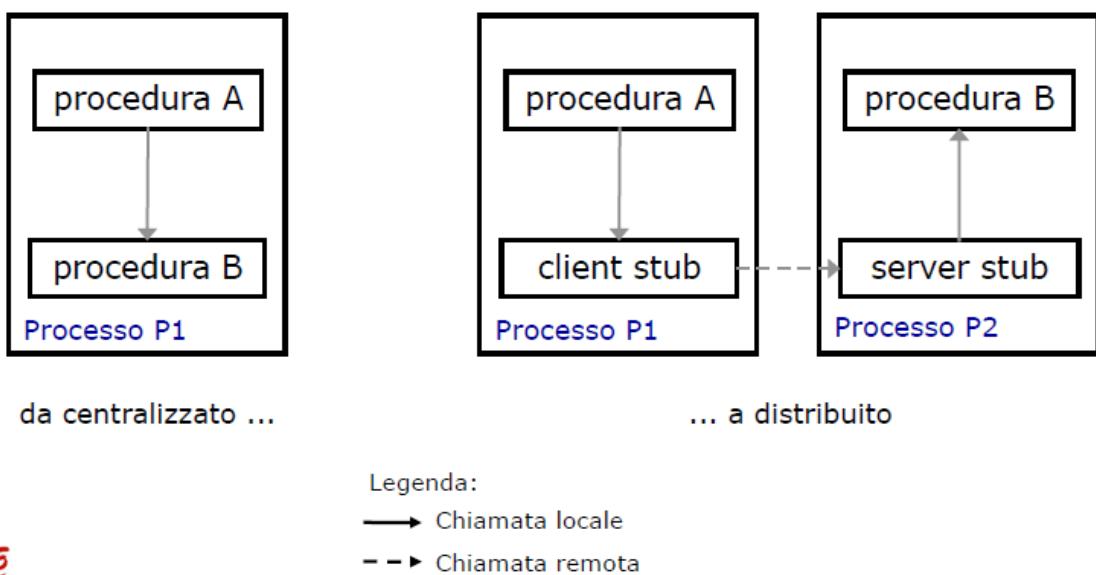
Tutti i metodi delle **servlet** sono supportati in REST, e Spring MVC gestisce tutti i metodi http;

# REMOTE PROCEDURE CALL

## PANORAMICA

Molti sistemi distribuiti si basano sullo scambio di messaggi tra processi. Tuttavia, le procedure **send** e **receive** non nascondono del tutto la comunicazione, il che è importante per ottenere la trasparenza all'accesso nei sistemi distribuiti.

Con l'utilizzo delle **chiamate a procedure remote RPC** invece, quando un processo sulla macchina A richiama una procedura sulla macchina B, il processo chiamante su A viene sospeso e ha luogo su B l'esecuzione della **procedura chiamata**. Le informazioni possono essere trasportate dal chiamante al chiamato attraverso i parametri e possono tornare indietro nel risultato della procedura. Nessun passaggio di parametri è visibile al programmatore.



## VANTAGGI

- Utilizzo di una semantica nota: chiamata di procedura;
- Sono facili da implementare: vicine al modello client-server;

## SVANTAGGI

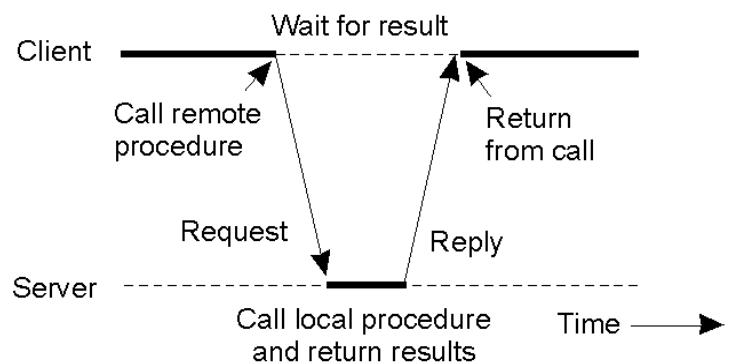
- Realizzate dal programmatore: tutto è esplicito;
  - ✓ [si può ovviare usando modelli a componenti più sofisticati]
- Sono statiche: scritte nel codice dai programmatori;
  - ✓ [si può ovviare usando **chiamate indirette**]
- Non c'è concorrenza: sono bloccanti;
  - ✓ [si può ovviare con un uso appropriato dei **thread** o **chiamate asincrone**]

## CLIENT E SERVER STUB

L'idea alla base delle RPC è di rendere una chiamata a procedura remota il più simile possibile a una chiamata locale. In altre parole vogliamo che le RPC siano **trasparenti**: la procedura chiamante **non deve essere consci** che la procedura chiamata è in esecuzione su una macchina diversa o viceversa.

Supponiamo che un programma necessiti di leggere dei dati da un file. Il programmatore mette una chiamata alla **read** nel codice per ottenere i dati. In un sistema tradizionale (a singolo processore), la routine **read** è estratta dal linker dalla libreria e inserita nel programma oggetto. In poche parole la procedura è una sorta di interfaccia tra il codice utente e il SO locale.

Quando la **read** è invece una **procedura remota**, ne viene messa nella libreria una versione diversa, chiamata **client stub**. A differenza dell'originale essa non richiede al sistema operativo di restituirlle soltanto i dati. Invece, impacchetta i parametri in un messaggio e ne richiede l'invio al server. (figura a fianco).



Dopo la chiamata alla **send**, il **client stub** richiama la **receive** e si blocca finché non riceve la risposta.

Quando il messaggio raggiunge il server, il sistema operativo di quest'ultimo lo passa al **server stub**. Un **server stub** è l'equivalente lato server del client stub: è un pezzo di codice che trasforma le richieste in ingresso dalla rete in chiamate di procedura locali. In genere il **server stub** avrà chiamato la **receive** e sarà bloccato in attesa di messaggi in ingresso. Il **server stub** spacchetta i parametri del messaggio e quindi richiama la procedura nel solito modo (localmente). Dal punto di vista del server, è come se fosse davvero richiamato dal client: i parametri dell'indirizzo di ritorno sono tutti sullo stack di appartenenza e tutto sembra normale. Il server esegue quindi il suo compito e restituisce il risultato.

Quando il server stub riprende il controllo dopo che la chiamata è stata completata impacchetta il risultato (buffer) in un messaggio e richiama la **send** per restituirlo al client. Quando il messaggio ritorna al client, il SO vede che è indirizzato al processo client (o in realtà al client stub ma il SO non vede la differenza). Il messaggio viene copiato nel buffer in attesa e il processo client sbloccato. Il client stub ispeziona il messaggio, spacchetta il risultato, lo copia al chiamante e termina nel solito modo.

## IN SINTESI

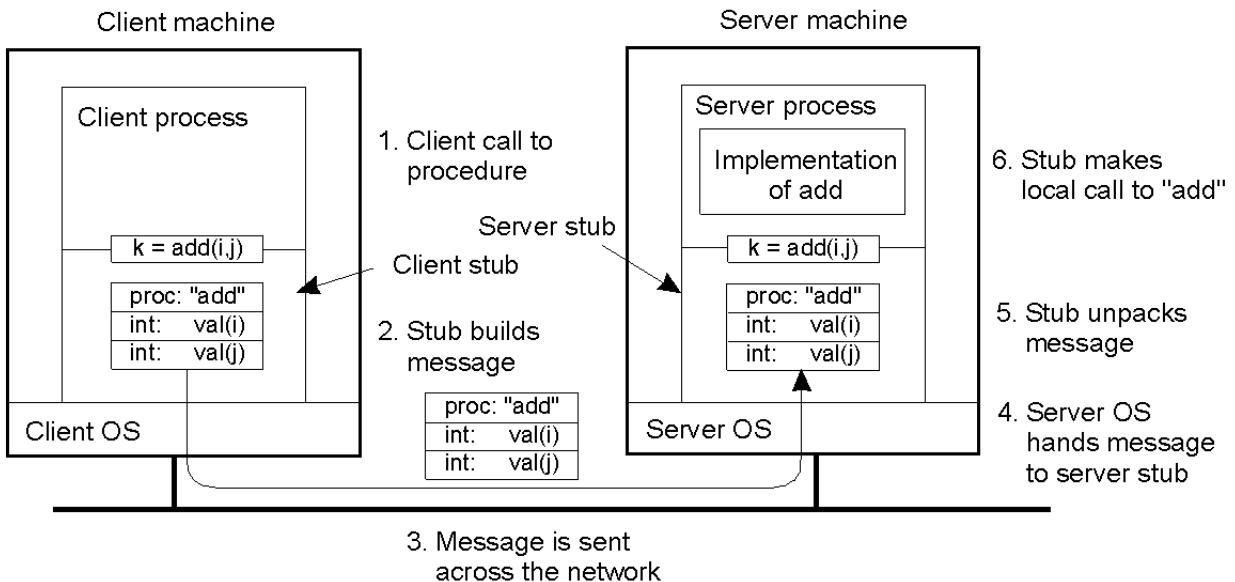
1. La procedura client richiama il **client stub** nel modo normale;
2. Il **client stub** costruisce il messaggio e richiama il sistema operativo locale;
3. Il SO del client invia il messaggio al SO remoto;
4. Il SO remoto passa il messaggio al **server stub**;
5. Il **server stub** spacchetta i parametri e richiama il server;
6. Il server esegue il lavoro e restituisce il risultato allo **stub**;
7. Il **server stub** lo impacchetta in un messaggio e richiama il suo SO;
8. Il SO del client passa il messaggio al **client stub**;
9. Lo **stub** spacchetta il risultato e lo restituisce al client;

## PASSAGGIO DI PARAMETRI

La funzione del client stub è di prendere i propri parametri, impacchettarli in un messaggio e inviarli al server stub. Di seguito analizziamo alcune questioni relative al passaggio di parametri;

### PER VALORE

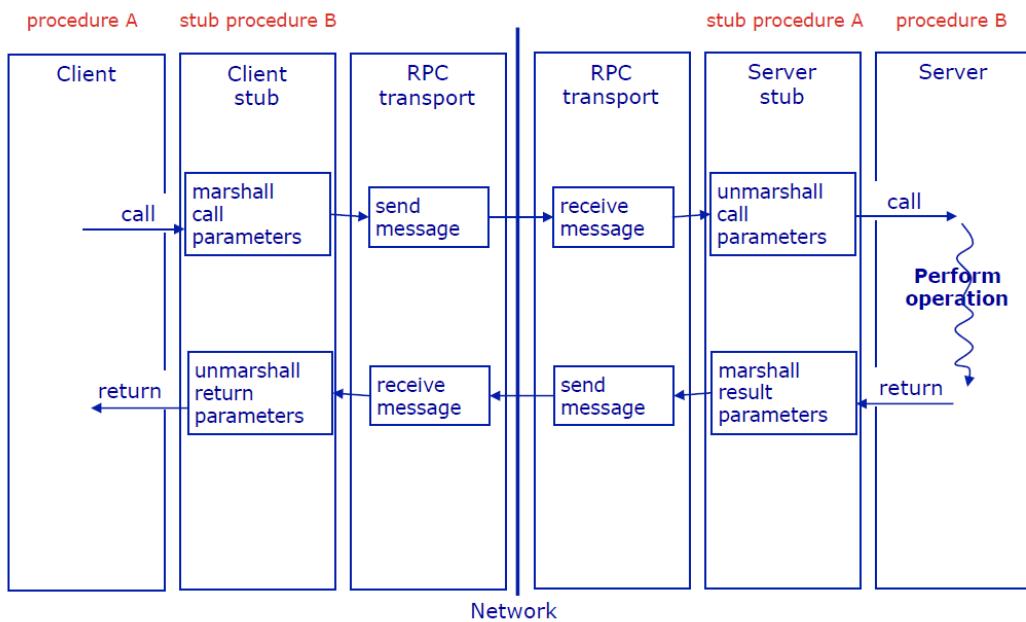
L'operazione di impacchettare i parametri in un messaggio è detta **marshaling dei parametri**. Come esempio consideriamo una procedura remota, *somma(i, j)*, che prenda due parametri interi i e j e restituisca come risultato la loro somma aritmetica.



La chiamata alla procedura *somma* è mostrata nella parte sinistra della figura sovrastante (processo client). Il client stub prende i suoi due parametri e li mette in un messaggio come indicato. Anche il nome o il numero della procedura da vengono messi nel messaggio, in quanto il server potrebbe supportare diverse chiamate e gli va indicato quale sia richiesta.

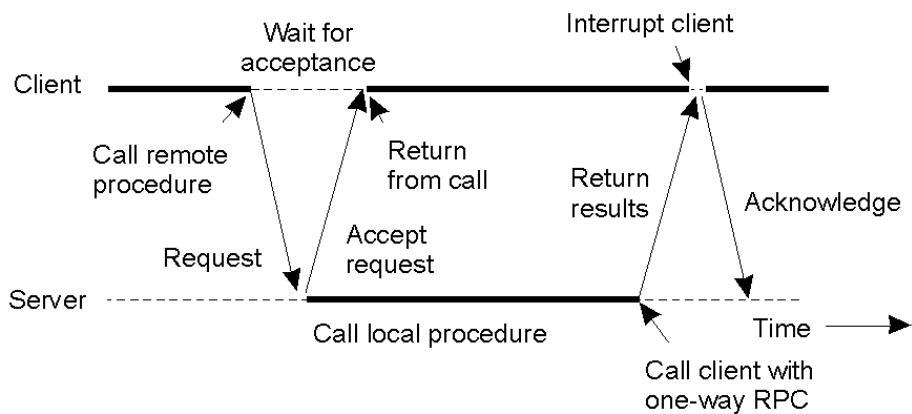
Quando il messaggio raggiunge il server, lo stub lo esamina per vedere quale procedura sia richiesta e quindi fa la chiamata appropriata. Quando il server ha terminato, il server stub riprende il controllo, preleva il risultato restituito dal server e lo impacchetta in un messaggio. Questo messaggio viene restituito al client stub, che lo spacchetta e restituisce il valore alla procedura client in attesa.

## ESECUZIONE DI UNA RPC



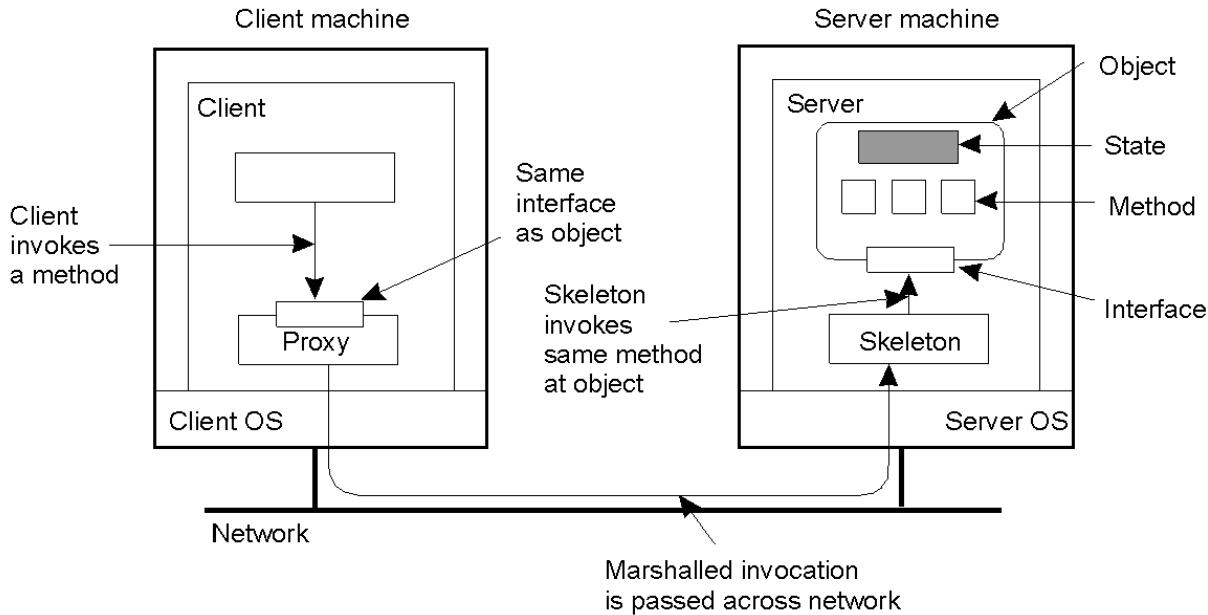
## RPC ASINCRONE

Come nelle chiamate a procedure convenzionali, quando un client richiama una procedura remota, quest'ultimo si blocca finché non riceve una risposta. Questo rigoroso comportamento richiesta-risposta non è necessario in assenza di un risultato da restituire e causa solo il blocco del client quando avrebbe potuto invece continuare l'elaborazione e fare lavoro utile semplicemente dopo aver richiesto la procedura remota da richiamare. In questo caso il client utilizza una **chimata remota asincrona**.



# RMI: REMOTE METHOD INVOCATION

## ARCHITETTURA DI RIFERIMENTO



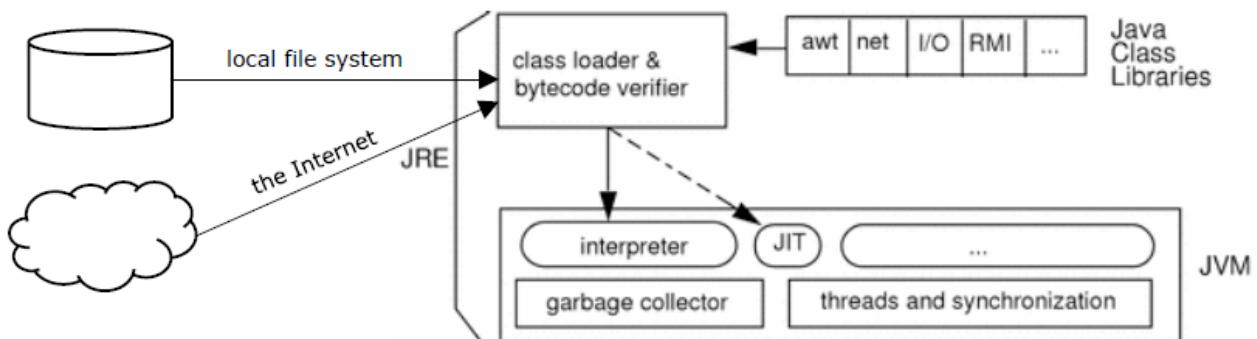
RMI ha come riferimento il modello **client-server**, con middleware o **stub** (proxy, skeleton) che forniscono i servizi per la distribuzione, e utilizza il **marshalling**. Il messaggio ha un proprio **stato** e un insieme di **metodi** che possono essere invocati, che formano un'interfaccia implementata da una classe.

## PANORAMICA

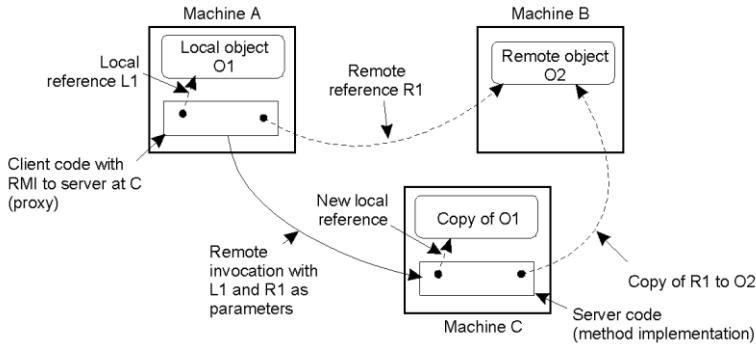
RMI è un **middleware** che fornisce servizi di garbage collector, class loader, security manager (filtrà le richieste di I/O), attivazione automatica degli oggetti e **multithreading**.

L' RMI supporta l'invocazione di metodi tra oggetti in macchine virtuali distinte con interfaccia Java, dunque vengono passati e ritornati oggetti java e le classi vengono caricate dinamicamente. E' basato sulla **portabilità del bytecode** e sulla macchina virtuale, estendendo l'approccio OO al distribuito (dalla virtual machine alla rete).

## AMBIENTE JAVA



La classe **loader** si occupa di caricare le classi (files .class) da librerie, file Systems locali, o da domini di rete. L'azione di caricamento avviene in modo dinamico quando viene lanciato un comando di **new** o quando viene trasferito un oggetto RMI.



**JAVA RMI** è simile a RPC perché gestisce i parametri per valore, ma è a oggetti e consente anche il passaggio per reference; definisce **stub** specifici per ogni oggetto, usando invocazioni statiche o dinamiche che includono informazioni logiche sull'identità dell'oggetto e del metodo. Il passaggio avviene per reference e non puntatori perché non è possibile conoscere le allocazioni di memoria di un altro processo.

Gli oggetti sono locali o remoti, passati tramite reference remota, valore o copia; ogni macchina crea una reference locale nel proprio spazio degli indirizzi. Le macchine accedono agli oggetti inviati sulla porta appropriata. Le copie devono contenere lo stato dell'oggetto.

### SERIALIZZAZIONE DI OGGETTI

Gli oggetti non remoti vengono passati per valore se **serializzabili**: la serializzazione è essenziale per memorizzare e ricostruire lo stato degli oggetti, per definirli e trasferirli via rete. La **serializzazione** rappresenta lo stato di un oggetto come stream di byte, il che permette di passare come parametro lo stato di un oggetto. Il meccanismo di loading dinamico di java permette di passare solo le informazioni essenziali sullo stato: la descrizione della classe può essere caricata a parte.

Il metodo `writeObject(Object obj)` attraversa tutti i riferimenti contenuti in obj per costruire una rappresentazione completa del grafo

**Esempio:**

```

// Serialize today's date to a file.
FileOutputStream f = new FileOutputStream("tmp");
ObjectOutput s = new ObjectOutputStream(f);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();

// Deserialize a string and date from a file.
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();

```

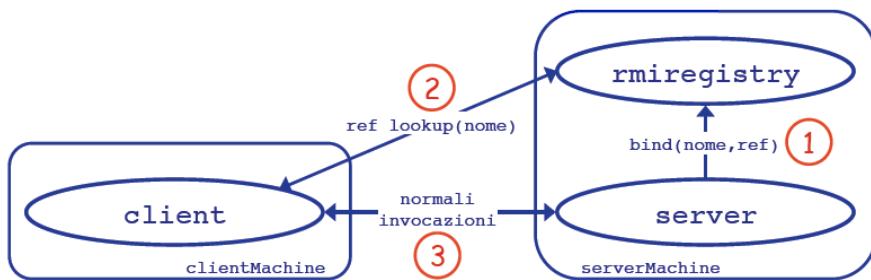
I tipi base (*boolean, byte, character, class, double, float, integer, long, ...*) sono **serializzabili** in modo nativo. Tuttavia è possibile creare classi serializzabili implementando l'interfaccia **Serializable**, ridefinendo i metodi

- `private void writeObject(java.io.ObjectOutputStream out) throws IOException;`
- `private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;`



## IDENTIFICAZIONE DEGLI OGGETTI

RMI definisce un **rmiregistry**, presente su ogni macchina che ospita servizi remoti, convenzionalmente sulla porta 1099; RMI attiva un servizio di ascolto per il servizio remoto ospitato da tale macchina.



La classe **naming** da accesso diretto alle funzionalità del **registry**, con metodi **static**, e parametri di tipo stringa in formato **URL** riferiti al registry e all'oggetto remoto considerato

`[//<host_name>[:<name_service_port>]/]<service_name>`

- Default host : localhost
- Default port: 1099

## Metodi

```
public static Remote lookup(String name)
    throws NotBoundException, MalformedURLException, RemoteException
    - Restituisce un riferimento, uno stub, all'oggetto associato al nome specificato.

public static void bind(String name, Remote obj)
    throws AlreadyBoundException, MalformedURLException,
    RemoteException
    - Collega (bind) il nome specificato all'oggetto remoto.

public static void rebind(String name, Remote obj)
    throws RemoteException, MalformedURLException
    - Collega (bind) il nome specificato all'oggetto remoto,
    cancellando i collegamenti esistenti.
```

```
public static String[] list(String name)
    throws RemoteException, MalformedURLException
- Restituisce i nomi (in formato URL) degli oggetti del registry.
```

```
public static void unbind(String name)
    throws RemoteException, NotBoundException, MalformedURLException
- Distrugge il collegamento (bind) al nome specificato.
```

### Parametri

name - nome in formato URL  
 obj - reference per un oggetto remoto (solitamente uno stub)

### Eccezioni

**AlreadyBoundException** - se il nome è già associato

**NotBoundException** - se il nome non è attualmente associato

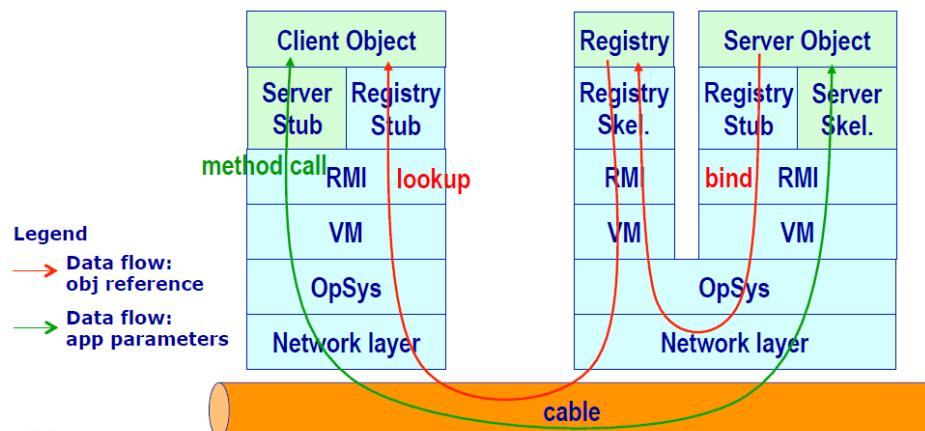
**RemoteException** - se il **registry** non può essere contattato

**AccessException** - se l'operazione non è permessa  
 (se originario di un non-local host, per esempio)

**MalformedURLException** - errata formattazione URL del nome

### ARCHITETTURA IN DETTAGLIO

Il server pubblica il reference e il nome dell'oggetto remoto nel **registry** con **bind**, il client ottiene il reference all'oggetto con **lookup** e accede dunque all'oggetto remoto



### Risoluzione dei problemi fondamentali con RMI

- Identificazione della controparte (**naming**) con il nome dell'host e dell'oggetto, l'RMI registry contiene l'elenco degli oggetti;
- Accesso alla controparte (**access point**), tramite reference ottenuto con il RMI registry e nomi locali per scrivere programmi;
- Comunicazione 1 (**data format**), oggetti Java serializzati o tramite reference;
- Comunicazione 2 (**data semantics, trasparenza**), classi e metodi Java e serializzazione;

Il livello di **trasparenza** è intermedio, è necessario conoscere la posizione del registry e il nome dell'host, ma l'utente non può distinguere tra oggetto locale o remoto perché la sintassi utilizzata è la stessa.

## CREARE UN OGGETTO REMOTO

Java definisce un'interfaccia per implementare oggetti remoti:

```
interface java.rmi.Remote
```

Per creare una classe remota **server** bisogna:

1. Definire l'interfaccia della classe remota
2. Implementare la nuova interfaccia
3. Implementare un server che crei e registri l'oggetto al registry

Il server crea un oggetto locale che realizza il servizio desiderato e lo registra presso l'RMI registratore sullo stesso host con un nome pubblico

```
MyRemoteObject myObj = new MyRemoteObject();
Registry localRegistry = LocateRegistry().getRegistry();
localRegistry.bind("publicName", myObj);
```

**Esempio:** realizzazione di una semplice applicazione: *echo*; riceve una sequenza di bytes dal client e la restituisce; il client contatta a turno 5 oggetti remoti, istanze di una classe server che fa la *echo*

Quindi per prima cosa definiamo l'interfaccia della classe remota (1.), estendendo l'interfaccia Remote

```
1. public interface Echo extends java.rmi.Remote {
2.     public String call(String message)
3.         throws java.rmi.RemoteException;
4. }
```

Il metodo call riceve il messaggio che verrà restituito

**Nota:** ogni metodo deve lanciare java.rmi.RemoteException

Implementiamo poi la nuova interfaccia (2.), si estende la classe UnicastRemoteObject

```
class java.rmi.server.RemoteObject
    implements java.rmi.Remote, java.io.Serializable)
class java.rmi.server.RemoteServer
    class java.rmi.activation.Activatable
        class java.rmi.server.UnicastRemoteObject
```

Implementiamo quindi il server che offre il servizio (3.)

```
1. import java.rmi.server.UnicastRemoteObject;
2. import java.rmi.Naming;
3. import java.rmi.RemoteException;
4. import java.rmi.MalformedURLException;
5. import java.rmi.registry.LocateRegistry;

6. public class EchoImpl extends UnicastRemoteObject // obbligatorio
   implements Echo
7. {
8.     private String name;
9. 
10.    // il costruttore è obbligatorio se estende UnicastRemoteObject
11.    // perché deve poter lanciare l'eccezione RemoteException
12.    public EchoImpl(String s) throws RemoteException {
13.        super();
14.        name = s;
15.    }
16. 
17.    // per controllo restituisce un messaggio composto per controllo
18.    public String call(String message) throws RemoteException {
19.        return name + " received: " + message;
20.    }
21. 
22.    public static void main(String args[]) {
23.        try {
24.            // start the rmiregistry, if already running remove the line
25.            LocateRegistry.createRegistry(1099);
26. 
27.            for (int i = 1; i <= 5; i++) {
28.                System.out.println("EchoImpl.main: create an EchoImpl");
29.                String name = "Echo" + i;
30.                EchoImpl echo = new EchoImpl(name);
31.                System.out.println("EchoImpl.main: bind it to name: "
32.                                  + name);
33.                Naming.rebind(name, echo);
34.            }
35.            System.out.println("Echo Server ready.");
36.        } catch (RemoteException | MalformedURLException e) {
37.            System.out.println("EchoImpl.main: an exception occurred: "
38.                              + e.getMessage());
39.            e.printStackTrace();
40.        }
41.    }
42. }
```

Non ci resta che implementare una classe client per usufruire del servizio echo

```
1. import java.rmi.Naming;
2. import java.rmi.NotBoundException;
3. import java.rmi.RemoteException;
4. import java.net.MalformedURLException;

5. public class EchoClient {
6.     public static void main(String args[]) {
7.         try {
8.             for (int i = 1; i <= 5; i++) {
9.                 String name = "Echo" + i;
10.                System.out.println("EchoClient: lookup " + name);
11.                Echo echo = (Echo) Naming.lookup(name); // explicit cast to class Echo
12.                // or Echo echo = (Echo) Naming.lookup("//localhost:1099/" + name);
13.                String message = echo.call("Hi #" + i);
14.                System.out.println("EchoClient: message from " + name);
15.                System.out.println("\t" + message + "\n");
16.            }
17.        } catch (RemoteException | NotBoundException | MalformedURLException e) {
18.            System.out.println("EchoClient: an exception occurred: " + e.getMessage());
19.            e.printStackTrace();
20.        }
21.        System.exit(0);
22.    }
23. }
```