Sistemi Distribuiti

Leonardo Valente

April 4, 2022

Contents

1 Introduzione

1.1 Che cosa è un sistema distribuito?

Ci possono essere diverse definizioni di "Sistema Distributo".

- Definiamo un sistema distribuito come un insieme di componenti Hardware e Software localizzati in una rete di computer che comunicano e coordinano le loro azioni solo passandosi messaggi.

- Un "Sistema Distribuito" è un insieme di elementi di computazione autonomi che appaiono all'utente come un singolo sistema coerente.

1.2 Caratteristiche

Gli "elementi di computazione autonoma" di un SD, anche chiamati "Nodi" sono i device hardware oppure i processi software. Il fatto un SD debba sembrare un singolo sistema all'utente implica il fatto che tra i nodi ci deve essere un sistema di collaborazione.
Quindi ogni nodo in quanto autonomo avrà la sua singola nozione di tempo. Non esiste un clock globable per tutti i nodi.
Questo porta quindi a problemi di sincronizzazione e di coordinamento.

La parola chiave resta sempre: **Trasparenza**

E' inevitabile il fatto che in qualsiasi momento solo parte del sistema fallirà. Nascondere questi fallimenti e il loro recupero è molto spesso difficile e in generale impossibile da nascondere.

Gestione della memoria?

- Non c'è memoria condivisa.
- Comunicazione via scambio di messaggi.
- Ogni componente conosce solo il proprio stato e può sondare lo stato degli altri

Gestione dell' esecuzione?

- Ogni componente è autonomo.
- Il coordinamento delle attività è importante per il funzionamento di un sistema formato da più componenti.

Gestione dell tempo?

- Non c'è un clock globale.
- Non c'è possibilità di scheduling globale.

Tipi di fallimenti

- Fallimenti indipendenti dei singoli nodi.
- Non c'è fallimento globale.

1.3 Gruppi

I gruppi possono essere **aperti** (tutti i nodi possono partecipare) o **chiusi** (solo membri selezionati possono partecipare).

2 Architetture Software

Una architettura software definisce la struttura del sistema, le interfacce tra i componenti e i pattern di interazione.
Ci possono essere diversi stili di architettura per un SD.

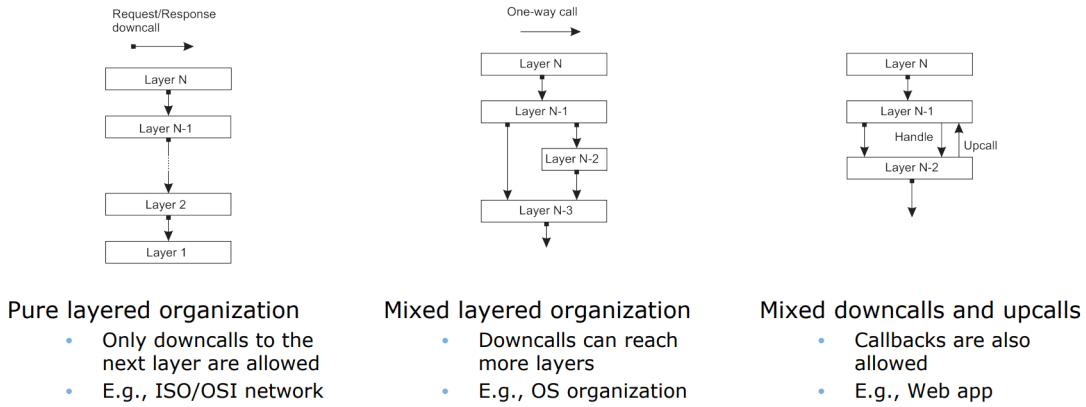
- **Architetture a strati** (layered)
  Ho un livello superiore che nasconde il lavoro effettuato dal livello sottostante.

- **Architetture a livelli**
  Applicazioni client server.

- **Architetture basate sugli oggetti**

- **Architetture basate su eventi**
  Applicazioni web dinamiche basate su callback (AJAX).

2.1 Tipi di layer



- Il primo caso è molto semplice. Abbiamo a disposizione N livelli di layer, ognuno che comunica solo ed esclusivamente con il livello sottostante.

- Il secondo caso invece potrebbe risultare un po più complicato. Per spiegarlo usiamo un semplice esempio: supponiamo che il nostro programma effettui una operazione di divisione per 0 (zero). Ovviamente sappiamo che ciò non è possibile, quindi passerà il compito di risolvere questa eccezione all'exception handler e successivamente tornerà a fare quello che doveva fare. Invece se effettuiamo una normale operazione che non genera eccezzioni ovviamente non dovrà fare la parte dell'exception handler.

- Per il terzo caso basti immaginare a come funziona AJAX o più in generale il funzionamento di una web app. Quindi chiamate al server con eventuale risposta e aggiornamento della UI.

2.2 Diversi tipi di Sistemi Distribuiti

E' importante differenziare i 3 tipi principali di SD.

- DOS (Distributed Operating System)
- NOS (Network Operating System)
- Middleware

Distributed Operating System

L'utente non è a conoscenza della molteplicità delle macchine che compongono il sistema.
I dati possono essere spostanti in modo intero o parziale, così come le operazioni di computazione.
Un processo può essere migrato interamente o in modo parziale su diversi siti, facendo così avremo un effetto di **Load balancing**, che ci permette di distribuire il carico di lavoro su più macchine, e di **Compuatation speedup** (i sottoprocessi possono essere eseguiti concorrentemente su più siti). Però il processo potrebbe aver bisogno di un determinato hardware (**Hardware preference**) oppure di un determinato software (**Software preference**). Avendo la possibilità di migrare il processo questo può essere possibile. Infine il processo può essere eseguito in modo remoto, invece che trasferire i dati in locale.

Network Operating System

L'utente è a conoscenza della molteplicità delle macchine che compongono il sistema.
NOS permette operazioni esplicite di comunicazione: **Socket**. (Comunicazione diretta tra processi.)
L'accesso alle risorse sulle varie macchine è effettuata in modo esplicito tramite:

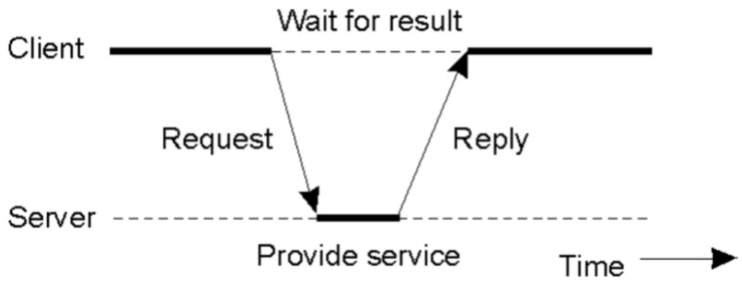
- Remote logging (telnet, ssh)
- Remote desktop
- FTP

Middleware

Il compito del middleware è quello di implementare i servizi per renderli trasparenti all'applicazione.

- Definisce e offre un modello di comunicazione che nasconde i dettagli dei messaggi passati.
- Definisce e offre un servizio automatico per il salvataggio dei dati (su file system o DB).
- Definisce e offre un modello persistente per garantire consistenza su operazioni di lettura e scrittura (di solito su DB).
- Definisce e offre modelli di protezione nell'accesso ai dati e servizi.

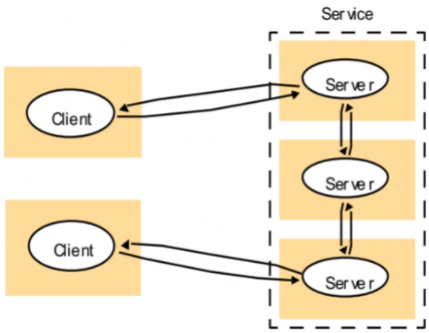
3 Il Modello Client-Server



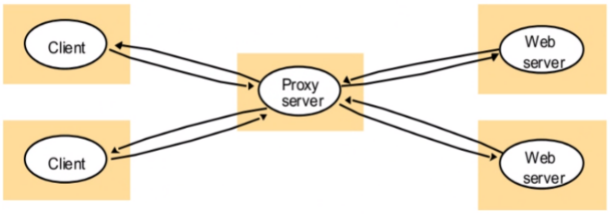
Di base questa architettura prevede che un **client** acceda ad un **server** con una richiesta e che il server risponda con un risultato.
Come possiamo notare dall'immagine, chiaramenta la richiesta con annessa risposta non è immediata, dovrà trascorrere un determinato quanto di tempo affinchè il server riesca a soddisfare la richiesta del client.

Ci possono essere diversi tipi di modelli.

- **Accesso a Server multipli**.
  Il client accede ad un server che a sua volta può accedere ad un altro server.



- **Accesso via Proxy**
  Il *Proxy* è un tipo di server che funge da intermediario per le richieste da parte di client alla ricerca di risorse su altri server.
  Il Server Proxy è molto utile per fornire l'anonimato durante la navigazione.



3.1 Problemi Fondamentali

In generale, ogni Sistema Distribuito deve aver a che fare con 4 problemi fondamentali:

- **Namig** (Identificare la controparte)
  Chi è la mia controparte? Dobbiamo assegnare dei nomi
- **Access point** (Accedere alla controparte)
  Come posso accedere ad una risorsa remota o un processo?
- **Protocol** (Comunicazione)
  Come posso scambiare messaggi? Bisogna mettersi d'accordo sul formato
- **Still an open issue** (Comunicazione)
  Come posso capire il contenuto di un messaggio?

3.2 Trasparenza

Il concetto fondamentale alla base di un buon Sistema Distribuito è, appunto, la trasparenza. Per cui, per esempio, il come un particolare dato venga rappresentato e la metodologia di accesso a quel dato sono *trasparenti* all'utente, non li vede. Così come anche la *location*. Non sappiamo dove quel dato risiede fisicamente.

Però, avere un **grado** di trasparenza troppo elevato potrebbe risultare eccessivo. Per esempio:

- Alcune *latenze di comunicazione* non possono essere nascoste.
- Nascondere i fallimenti del sistema e dei nodi è talvolta **impossibile**.
- Esporre le distribuzioni potrebbe essere alcune volte una buona pratica. Se un server non dovesse rispondere ad una chiamata per troppo tempo, bisogna riportare il fallimento per dare un feedback all'utente di cosa sta accadendo e di agire di conseguenza.
- Inoltre, la trasparenza completa ha un costo elevato (e.g. Mantenere le repliche di tutti i dati esattamente "up-to-date" con il master)

3.2.1 Information Hiding

L'Information Hiding è il principio che sta alla base dell'*Ingegneria del Software*.
E' importante fare una distinzione tra:

- **cosa** un servizio o un sistema mette a disposizione
  definisce l'*Application Programming Interface* (API) dei componenti o del sistema.
- **come** un servizio è stato implementato e distribuito
  definisce come il tool adatto per quel specifico problema.

Queste interfacce però devono essere sviluppate seguendo certi criteri, quindi devono mantenere una struttura che segua i principi prestabiliti, deve essere completa (mettere a disposizione tutto quello che serve) e neutrale.

3.3 Il concetto di Protocollo

Per poter capire le richieste e formulare le risposte i due processi devono concordare un **protocollo**.
I protocolli definiscono il **formato**, l'**ordine** di invio e di ricezione dei messaggi, il **tipo dei dati** e le **azioni** da eseguire quando si riceve un messaggio.
Alcuni esempi di protocolli sono:

- HTTP - HyperText Transfer Protocol
- FTP - File Transfer Protocol
- SMTP - Simple Mail Transfer Protocol

4 Le Socket

I processi sono i programmi in esecuzione, che sono aree di memoria gestite dal Sistema Operativo. Ogni processo comunica attraverso dei **canali**, che controllano i *flussi di dati in* ***entrata*** *e* ***uscita***.
Dall'esterno, ogni canale è identificato da un numero intero detto **porta**.

Le **socket** sono dei particolari canali per la comunicazione tra *processi* che non condividono memoria.
Per potersi connettere o inviare dati ad un processo A, un processo B deve conoscere la macchina (*host*) che esegue A e la porta a cui A è connesso. (*Well known port (?)*)

4.1 Servizi di trasporto di dati

TCP

Il servizio *TCP* è un protocollo **orientato alla connessione**, ovvero che il *client* invia al *server* una richiesta di connessione, e il server risponde di conseguenza.
TCP è famoso per il suo **trasporto affidabile**, tra la comunicazione tra processi.
Possiede anche un **controllo di flusso** (il mittente rallenta per non sommergere il ricevente) e un **controllo della congestione** (il mittente rallenta quando la rete è sovraccaricata).

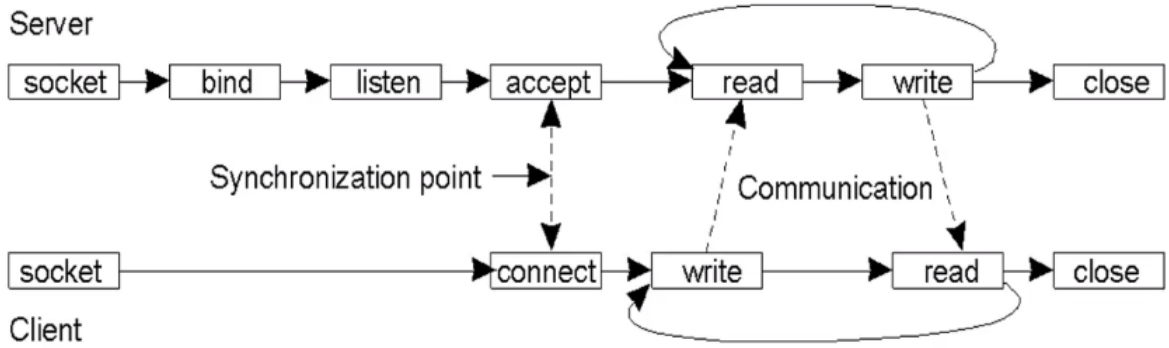
UDP

UDP è un protocollo che **non** garantisce la ricezione del messaggio, e in generale **non** possiede tutte le funzionalità di sicurezza del TCP.
Quindi, *perchè esiste UDP?*
UDP può essere molto utile per le applicazioni che tollerano perdite parziali (ad esempio un servizio di video streaming, anche se perdiamo qualche frame non ci accorgiamo nemmeno) a vantaggio delle prestazioni.

Le socket non sono altro che delle API per accedere ai servizi di trasporto TCP e UDP.

4.2 Comunicazione via socket

La comunicazione TCP/IP avviene attraverso **flussi di byte** dopo una **connessione esplicita**, tramite normali *System Call* di read/write.



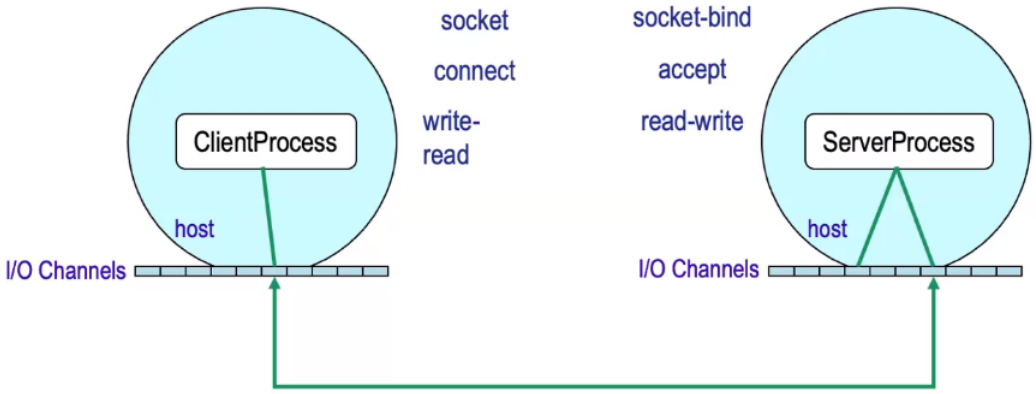
- **Socket**: viene creata un nuovo canale socket.
- **Bind**: viene associato un *local address (porta)* alla socket per l'identificazione del processo
- **Listen**: si mette in ascolto di nuove connessioni
- **Connect**: il client invia una richiesta di connessione al server
- **Accept**: accetta la connessione del client.
  Crea una nuova porta per gestire la **comunicazione** (dedicata).
- **Write**: invia dei dati attraverso il canale
- **Read**: riceve dei dati attraverso il canale
- **Close**: chiude la connessione

La **comunicazione** è un ciclo continuo di *read* e *write* tra il client e il server fino a che non viene chiusa la connessione.
Le socket trasportano degli *stream*, quindi non esiste il concetto di messaggio, è appunto un flusso continuo di dati.
Però la lettura e la scrittura avvengono per un numero arbitrario di byte.

Connect e **Accept** "bloccano" il sistema (rispettivamente il client e il server) fino a che non viene stabilita una connessione (vengono messi in attesa).

Perchè non viene creata solo una porta per la comunicazione ma ne vengono create molteplici?

Per gestire e identificare le varie connessioni che sono state create tra i client che si sono collegati al server.



4.3 Progettare una Applicazione con le Socket

- **Client**: L'architettura è più semplice di quella di un server:
  di solito è una applicazione che usa una socket anzichè un altro canale I/O
- **Server**: L'architettura prevede che:
  - venga creata una socket con una porta nota per accettare le richieste di connessione
  - entri in un ciclo infinito in cui alternare:
    * attesa/accettazione di una richiesta di connessione da un client
    * ciclo lettura-esecuzione
    * chiusura connessione

5 Architettura dei Server

I server possono essere:

- **Iterativi**: soddisfano una richiesta alla volta
- **Concorrenti a processo singolo**: simulano la presenza di un server dedicato
- **Concorrenti multi-processo**: creano server dedicati
- **Concorrenti multi-thread**: creano thread dedicati

5.1 Server Iterativi

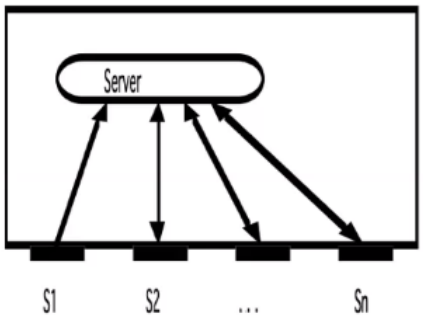
Al momento di una richiesta di connessione, il server crea una socket temporanea per stabilire una connessione diretta con il client.
Eventuali ultieriori richieste, verranno accodate alla porta nota per essere soddisfatte in seguito.

Questa architettura ha il **vantaggio** che è molto semplice da progettare, **però** viene servito un client alla volta, mettendo appunto in attesa gli altri.

5.2 Server Concorrenti

Un server concorrente può gestire più connessioni client.

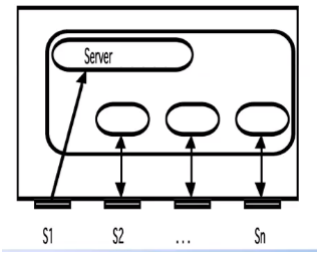
- **Monoprocesso**: Esistono delle funzioni (*select in C*) che vanno a selezionare i canali pronti all'uso.



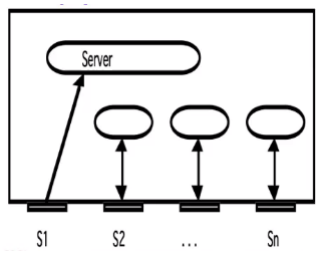
S1 è la socket per accettare le richieste di connessione, le altre sono le connessioni individuali.

Nei server monoprocesso, gli utenti condividono lo stesso spazio di lavoro, quindi adatto per le applicazioni cooperative che prevedono la modifica dello stato

- **Multi-thread**:
  Ho un processo singolo, e all'interno dei piccoli sotto processi (Thread) che simulano un processo a se

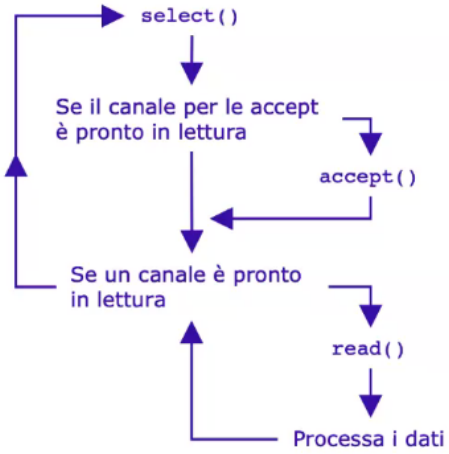


- **Multi-processo**: Ho degli effettivi processi clone, quindi veri e propri server "fisici", non simulati.



Nei server multiprocesso, ogni utente ha uno spazio di lavoro autonomo, quindi adatto per le applicazioni che non modificano lo stato del server, oppure per quelle applicazioni che prevedono una modifica solamente al proprio spazio di lavoro dedicato.

Per quanto riguarda la funzione di *select*, essa permette di gestire in modo non bloccante i diversi canali di I/O.
(Una istruzione *bloccante* è quando il sistema attende la conclusione dell'operazione richiesta prima di restituire il controllo al chiamante)



Struttura di un Server Concorrente

```
1.  create ServerSocketChannel;
2.  create Selector;
3.  set the ServerSocketChannel in non-blocking mode
4.  associate the ServerSocketChannel with the Selector;
5.  while(true) {
6.    waiting events from the Selector;
7.    event arrived;
8.    create keys;
9.    for each key created by Selector {
10.     check the type of request;
11.     isAcceptable:
12.       get the client SocketChannel;
13.       associate that SocketChannel with the Selector;
14.       record it for read/write operations
15.       continue;
16.    isReadable:
17.       get the client SocketChannel;
18.       read from the socket;
19.       process the read data
20.       continue;
21.    isWriteable:
22.       get the client SocketChannel;
23.       write on the socket;
24.       continue;
25.   }
26. }
```

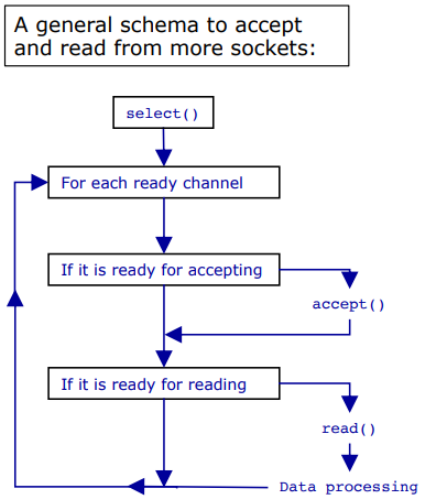


Figure 1: Un esempio di codice pratico lo puoi trovare al seguente link: `https://github.com/LeoRc01/Concurrent-Socket`

6 Architettura del Web

Il web supporta l'interazione tra Client e Server attraverso il protocollo HTTP.

- Il client è formato da *browser* oppure anche chiamato **User-agent**
- Il server è formato ad un *web server* o **HTTP Server**

6.1 Browser

Il browser è una applicazione che consente al client di navigare sul web.
La sua funzione primaria è quella di **interpretare** il codice con cui sono espresse le informazioni (pagine web) e **visualizzarle** sullo schermo.

6.2 Web page

Una pagina web è costituita da diversi risorse. Una risorsa è un file residente in un computer identificato da un URL, ovvero un indirizzo univoco che punta alla risorsa.
La maggior parte delle pagine web sono costituite da un file HTML.

6.3 Web server

Un web server è una applicazione che si occupa di gestire le risorse su un computer e di renderle disponibili al client

6.4 URL (Uniform Resource Locator)

Identifica un oggetto nella rete e specifica come interpretare i dati ricevuti attraverso il protocollo.
Un URL ha cinque componenti principali:

- Nome del protocollo
- Indirizzo dell'host
- Porta del processo
- Percorso dell'host
- Identificatore della risorsa

```
protocollo://indirizzo_IP[:porta]/cammino/risorsa
    1.              2.          3.      4.      5.
```

RFC 3986

```
ftp://www.adobe.com/dawnload/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
```

6.5 Linguaggi del web

I **dati** testuali sono espressi in linguaggi standard:

- HTML - per definire la struttura dei contenuti della pagina web.

  Un file HTML può contenere (nella maggior parte dei casi) dei contenuti scritti in CSS (gestisce la parte grafica e di "rendering")

- XML — JSON - focalizzato sui dati e la loro struttura

I dati possono anche essere non testuali (file di immagini, file di audio, file di video, ecc.) Essi vengono gestiti attraverso dei sistemi di *Encoding*.

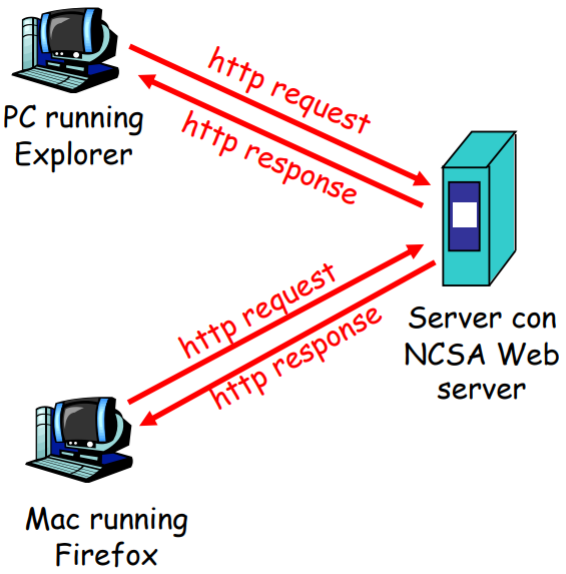
Esistono anche dei linguaggi di scripting (JavaScript, VBSCript, Java, ecc.) che permettono di arricchire la pagina web di funzionalità più complesse che magari non possono essere implementate esclusivamente attraverso HTML e CSS.

7 Message Oriented Comunication

7.1 Il protocollo HTTP

HTTP (o anche HTTPS) è il protocollo standard per la comunicazione, e lo scambio di risorse, tra client e server. HTTP usa TPC per la comunicazione.

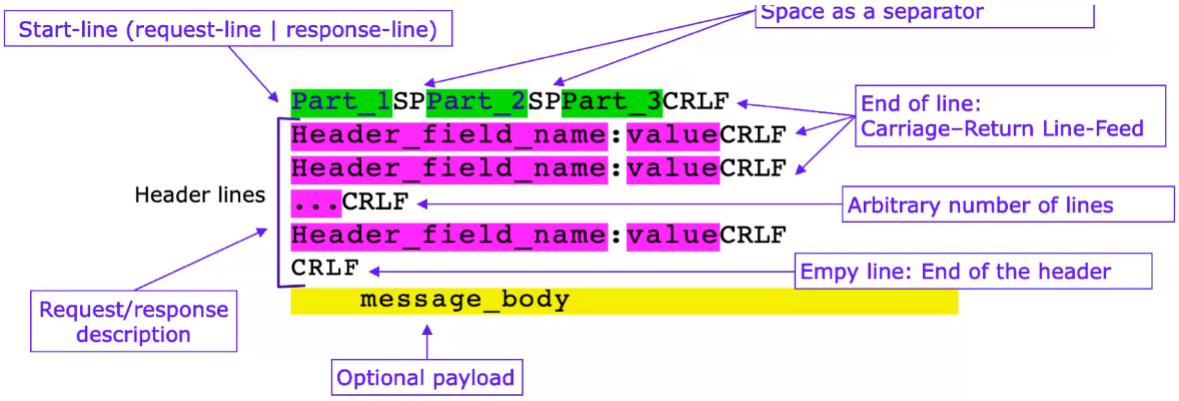
- Il client inizia una connessione TCP (crea una socket) verso il server sulla porta 80 (443 nel caso di HTTPS)
- Il server accetta la connessione TCP dal client
- Vengono scambiati i messaggi tra il browser e il Web-server



HTTP è un protocollo **stateless**, ovvero che non mantiene in memoria informazioni sul client.

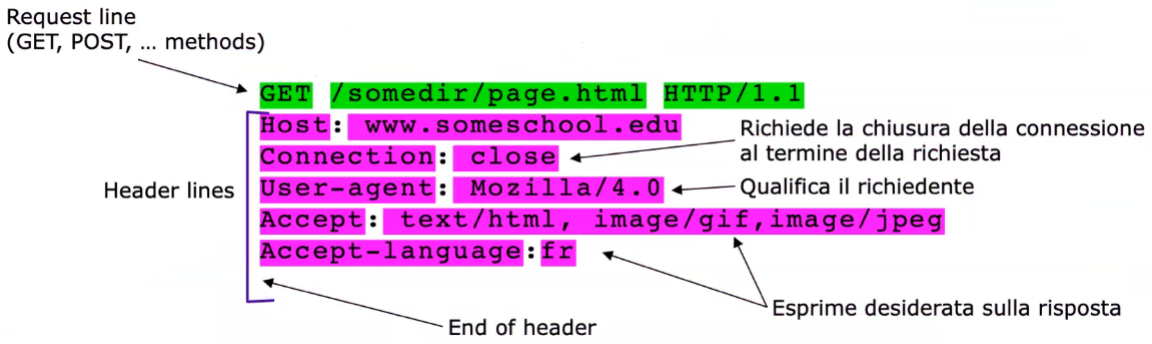
7.2 Formato dei messaggi HTTP

Formato **generale** di un messaggio HTTP:



La prima parte in verde è **obbligatoria**.
La parte gialla è il **payload** (il contenuto del messaggio).

Formato di una richiesta HTTP:



- **GET** - metodo della richiesta
- **/somedir/page.html** - percorso della risorsa
- **HTTP/1.1** - versione del protocollo
- **Host: www.someschool.edu** - URL del server
- **Connection: close** - cosa fare al termine del messaggio (in questo caso chiudere la connessione)
- **User-Agent: Mozilla/4.0** - il browser utilizzato dal client

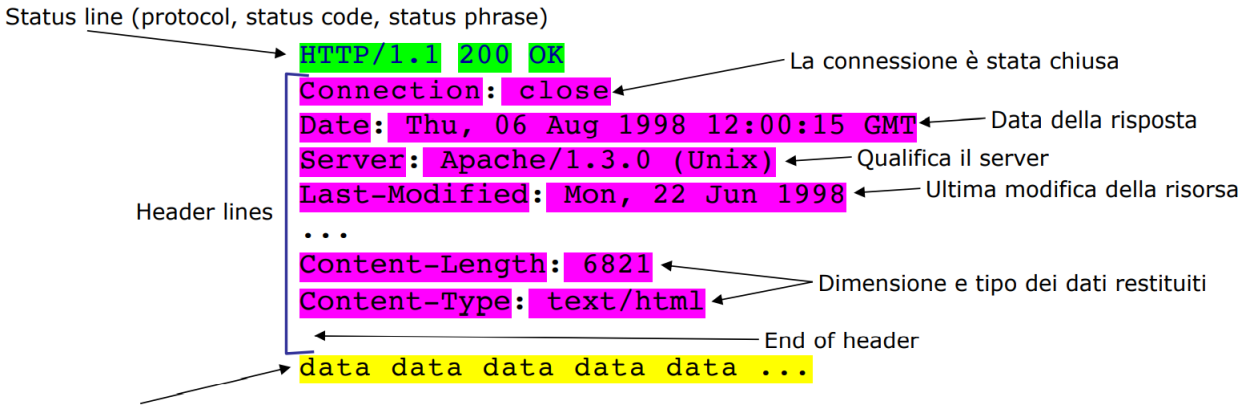
- **Accept: text/html, image/gif, image/jpeg** - che tipi di file sono accettati
- **Accept-Language: fr** - linguaggio della risorsa richiesta. (Se hai la risorsa francese, mandamela in francese).

Metodi di richiesta

- **GET**
  - Restituisce la risorsa richiesta
  - Include eventuali parametri in coda alla URL della risorsa
  - L'esecuzione non ha effetti sul server
  - Esiste anche il **GET Condizionale**, ovvero non inviare oggetti che il client ha già in cache
- **POST**
  - Comunica dei dati da elaborare lato server
  - L'input segue come documento autonomo
  - I parametri passati non vengono mostrati esplicitamente
- **HEAD**
  - Simile a GET, ma viene restituito solo il header della pagina web
  - Usata per il **debugging**

		cache	safe	idempotent
OPTIONS	represents a request for information about the communication options available on the request/response chain identified by the Request-URI			✓
GET	means retrieve whatever information (in the form of an entity) is identified by the Request-URI	✓	✓	
HEAD	identical to GET except that the server MUST NOT return a message-body in the response	✓	✓	
POST	is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line			
PUT	requests that the enclosed entity be stored under the supplied Request-URI			✓
DELETE	requests that the origin server delete the resource identified by the Request-URI			✓
TRACE	is used to invoke a remote, application-layer loop- back of the request message			✓

Formato di una response HTTP:



Perchè sono state inserite informazioni sulla dimensione e sul tipo dei dati restituiti?

- **Lenght** - mi serve per vedere fino a quando devo leggere
- **Type** - per far capire al client come deve interpretare la risposta

7.3 Codici di risposta

`1xx` (Informational):	Request received; server is continuing the process.
`2xx` (Success):	The request was successfully received, understood, accepted and serviced.
`3xx` (Redirection):	Further action must be taken in order to complete the request.
`4xx` (Client Error):	The request contains bad syntax or cannot be understood.
`5xx` (Server Error):	The server failed to fulfill an apparently valid request.

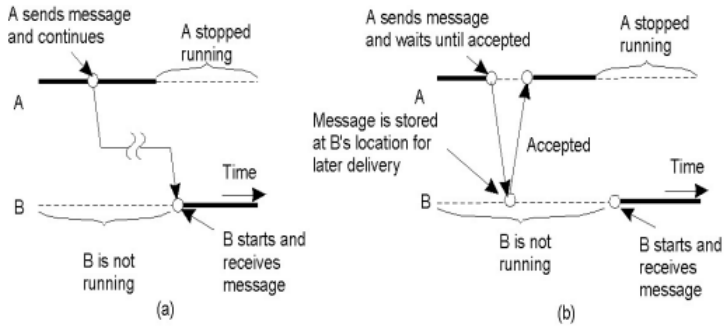
Alcuni esempi più comuni

`200 OK`	Successo, se richiesto l'oggetto è contenuto nel messaggio
`301 Moved Permanently`	L'oggetto richiesto è stato spostato. Il nuovo indirizzo è specificato nel'header (`Location: …`)
`400 Bad Request`	Richiesta incomprensibile al server
`404 Not Found`	Il documento non è stato trovato sul server
`505 HTTP Version Not Supported`	

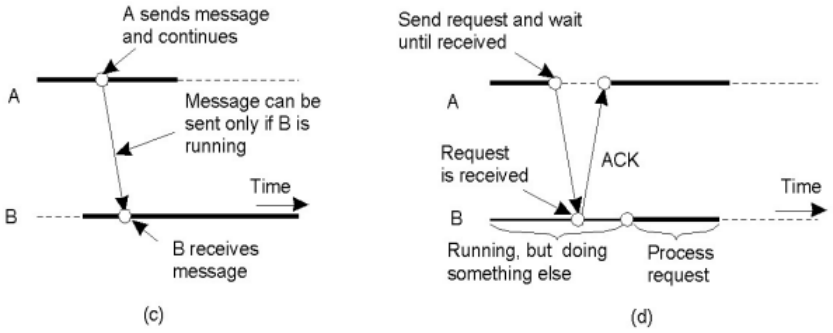
7.4 Tipi di comunicazione

- Comunicazione **asincrona o sincrona**
- Comunicazione **transiente** (se il destinatario non è connesso, i dati vengono scaricati)
- Comunicazione **persistente** (il middleware memorizza i dati fino alla consegna del messaggio al destinatario)

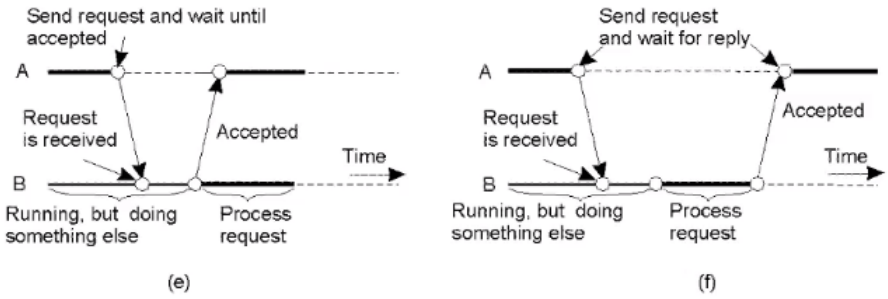
a) Persistent asynchronous communication
b) Persistent synchronous communication



c) Transient asynchronous communication
d) Receipt-based transient synchronous communication

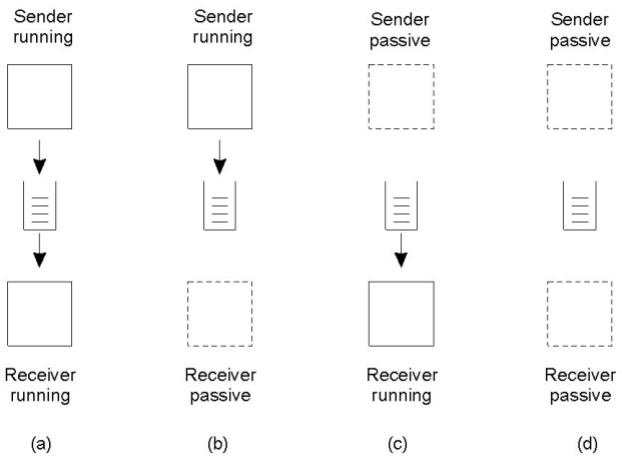


e) Delivery-based transient synchronous communication at message delivery
f) Response-based transient synchronous communication



7.5 Message Queing

Offre uno spazio di storage intermedio per i messaggi, senza che ne il client o il server siano attivi durante la trasmissione.
Esistono 4 tipi di comunicazione: 4 tipi di primitive:



- **Put** - appende un messaggio ad una coda specifica
- **Get** - Blocca fino a che la coda specifica è non vuota, e rimuove il primo messaggio
- **Poll** - Rimuove il primo messaggio ma non blocca
- **Notify** - installa un handler che viene chiamato quando un messaggio è inserito dentro una coda

8 Applicazioni Web

La computazione dei dati avviene lato server e può avvenire tramite **programmi compilati** o **script interpretati**.

Nel caso di **programmi compilati**, il web server si limita ad invocare un eseguibile.

Nel caso di **script** il web server ha un suo **engine** (motore) in grado di interpretare il linguaggio di scripting usato. (PHP, Java, Perl, Python, NodeJS)

CGI (Common Gateway Interface)

CGI è un protocollo che permette al server di:

- attivare un programma
- passargli le ricehiste e i parametri provenienti dal client
- recuperare la risposta

Questo protocollo viene gestito dall'interprete per il linguaggio usato (PVM Interpreter, JVM Interpreter, PHP Interpreter)

8.1 Java Servlet

Sono piccole applicazione Java residenti sul **server**. Essa viene gestita in modo automatico da un **container** o **engine**.

Ha una interfaccia che definisc il set di metodi (ri)definibili.

Siccome risiedono sul server, le servlet mantengono uno stato e consentono l'interazione con un'altra servlet. (sono oggetti Java alla fine)

HTTP però non prevede persistenza (è stateless), per cui mantenere lo stato della conversazione è un compito dell'applicazione (se vuole):

- Cookies
  - Le informazioni vengono memorizzate sul client
  - Permettono di gestire le sessioni
- HTTPSession

Una servlet viene creata dal container quando viene creata dal container. Essa viene condivisa da tutti i client, però ogni richiesta genera un Thread che esegue la *doXXX* appropriata.
Viene invece distrutta quando:

- non ci sono più servizi in esecuzione
- dopo un quanto di tempo predefinito (timeout)

8.2 Pattern MVC

Il pattern MVC (Model View Controller) ha lo scopo di separare

- I dati e i metodi per manipolarli (Model)
- La presentazione, la GUI (View)
- Il coordinamento dell'interazione tra interfaccia e i dati (Controller)

Vantaggi

MVC offre una chiara separazione tra i vari layer di sviluppo di una applicazione (business, presentazione e dati) Ogni componente ha una responsabilità ben definita e ogni parte può essere affidata ad un esporto. Questo modello permette anche di creare un codice più ordinato, quindi più facile da mantenere e da aggiornare.

Svantaggi

- Inefficenza nel passaggio dei dati alla view (mediazione del controller)
- Aumento della complessità dovuta alla concorrenza (è un sistema distribuito, più "attori" da coordinare e mettere assieme)

9 Servizi e SOAP

Service Oriented Architecture (SOA) è uno stile architetturale per costruire *applicazioni distribuite*, dove l'accesso al servizio può essere fornito tramite l'internet.

L'elemento principale di questo sistema è il **servizio**. Esso fornisce **features** e **funzioni** a client *indipendenti*. Ogni servizio dovrebbe:

- fornire una *descrizione* da scoprire e selezionare
- fornire accesso tramite protocolli standard
- indirizzare i bisogni di business del cliente e i requisiti di dominio (funzionali e non-funzionali)

9.1 Web Service

Un *Web Service* è una **applicazione software**

- identificata da un URI
- le quali interfacce e "bindings" sono capaci di essere **definite, descritte e scoperte**
- supporta interazioni dirette con altri software usando *messaggi* attraverso protocolli basati sull'internet

Una definizione più informale ma diretta è:
" *Un web service è un software indipendente che può essere scoperto e invocato da altri sistemi software attraverso l'internet.*"

9.2 Service Model: SLA

SLA (Service Level Agreement) è un contratto definito tra il provider e l'utente dei servizi.
Questo contratto definisce il requisiti **non-funzionali** che il servizio deve *garantire*.

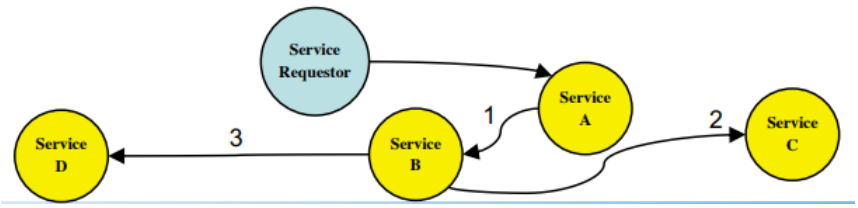
Response Time	Max	0.25 s
	Mean	0.10 s
Availability	Min	99.99%
Cost	Mean	0.02 €

Cost, Availability e Response Time vengono chiamati **SLO** (Service Level Objectives).

9.3 Composizione di Servizi

Una composizione consiste in una serie di servizi che sono interconnessi tra loro, che a turno potrebbero essere usati come un nuovo servizio in altre composizioni. Ovviamente ci deve essere *compatibilità* tra i servizi per avere una composizione di successo.

- **Orchestrazione** - descrive come i servizi devono interagire tra di loro a livello di messaggistica, includendo la *business logic* e *l'ordine di esecuzione*
- **Coreogragia** - descrive la sequenza di messaggi che potrebbero coinvolgere più parti e più fonti coinvolte nel processo dalla prospettiva di tutte le parti



9.4 WSDL

WSDL (Web Service Description Language) è un linguaggio **XML-Based** per descrivere i servizi e come accederci.
Descrive:

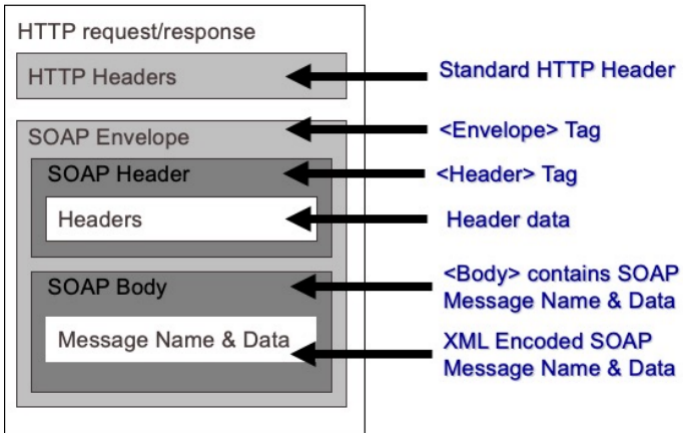
- Interfacce - informazioni che descrivono tutte le operazioni pubbliche
- Tipi di dato - dichiarazione per tutti i messaggi di richiesta e di risposta
- Binding - informazioni sul protocollo di trasporto
- Address - informazioni sulla località del servizio (URI)

9.5 SOAP

SOAP è un protocollo basato su XML per permettere alle componenti dei software e delle applicazioni di comunicare usando protoclliu standard dell'internet (e.g HTTP)

- SOAP è un protocollo stateless e **one-way**
- Ignora la semantica del messaggio trasferito
- L'interazione tra i due siti deve essere codificata all'interno del documento SOAP

Questo protocollo non è legato a nessuno linguaggio di programmazione o piattaforma, è però semplice e estendibile.



SOAP però non è più molto usanto in quanto troppo verboso (rispetto ad esempio a JSON).

10 REST

REpresentational **S**tate **T**ransfer, è uno stile architetturale per i sistemi distribuiti.

- Le risorse sono definite tramite URI (/baseURL/resourceType/id) e sono manipolate tramite le loro rappresentazioni
- Ci possono essere più tipi di rappresentazione per una risorsa (XML, JSON, HTML)
- La comunicazione avviene tramite messaggi, che sono auto-descrittivi e stateless
- Lo stato di una applicazione avviene attraverso la manipolazione delle risorse
- Hypermedia è il modo per controllare il comportamento dell'applicazione

Risorse

Una risorsa è un qualsiasi tipo di informazione che può avere un nome: documenti, immagini, servizi...
Le risorse hanno uno **stato** (Luce - accesa/spenta) che può cambiare nel tempo, tramite (di solito) una interazione con l'utente.
Le risorse hanno un identificatore univoco (unimib.it/{matricola}/pianoStudio)

Le risorse per essere acquisite e rappresentate richiedono una architettura *client - server*. L'utente ha anche la possibilità di manipolare una risorsa, per esempio: uno studente può manipolare il proprio piano di studi, aggiungendo o rimuovendo i corsi. Tendenzialmente, la rappresentazione ritornata dal server dovrebbe collegarsio a un'altra rappresentazione/azione. (Hypermedia)

Le interazioni devono essere **stateless**, quindi non devono usufruire di informazioni di contesto salvate nel server. Viene quindi usato un sistema di **caching** che serve anche per ridurre la latenza e il traffico sulla rete.

10.1 Costruzione di un servizio REST

- Trovare tutti i **nomi**
- Definire i formati
- Scegliere le operazioni
- Evidenziare i codici di stato (soprattuto in caso di errore)

Trovare tutti i nomi

Tutte le risorse all'interno di un servizio REST sono un URI, che di per se deve essere **descrittivo**. (dato un URI, devo riuscire a capire cosa mi restituisce)
Bisogna usare le **path variables** per creare una gerarchia delle informazioni e per tenerle più ordinate. (/expenses/pending/{id})
Si possono usare le **query variables** per dare in input all'algoritmo determinati dati per ottenere un certo tipo di informazione. (/search?approved=false)
E' importante anche non mostrare le informazioni della piattaforma usata per creare il servizio (e.g /expensese.*php*)

Definire i formati

Il consiglio è quello di non creare dei tipi di rappresentazione "custom", quindi usare quelli standard (tipo JSON, XML, ecc...) Un client dovrebbe avere le possibilità di ottenere, modificare o eliminare un documento e il server deve scartare qualsiasi altro tipo di richiesta.

Scegliere le operazioni

Per la maggior parte delle applicazioni, i metodi di base dell'HTTP sono sufficenti:

- GET - richiedere una risorsa
- POST - aggiungere nuove risorse
- PUT - trasferisce una risorsa al server
- DELETE - elimina una risorsa

PUT vs POST:

- POST quando il server sceglie l'URI

- PUT quando il client sceglie l'URI

POST fa qualcosa e riceve qualcosa, che non sempre può essere necessario, quindi crea un **overload** di informazioni.

Evidenziare i codici di stato

Bisogna dare un feedback all'utente dopo aver effettuato una operazione. Quindi se il client richiede una risorsa non esistente, riceverà come risposta un determinato codice e seguirà un feedback (attraverso la UI) per l'utente spiegando che la risorsa non esiste.
Questo concetto va applicato anche per gli errori e i successi di una richiesta.

11 HTML+(DOM)+CSS

11.1 Linguaggi di Markup

Un linguaggio di Markup è utilizzato per *annotare un documento* in modo tale che l'annotazione sia *sintatticamente distinguibile* dal testo.
Le annotazioni possono avere diverse finalità:

- Presentazione (definiscono come visualizzare il testo)
- Precedurali (definiscono istruzioni per programmi che elaborino il testo al quale sono associate)
- Descrittive (metadati, possono servire per il browser per identificare alcune informazioni, etichettano semplicemente parti del testo,)

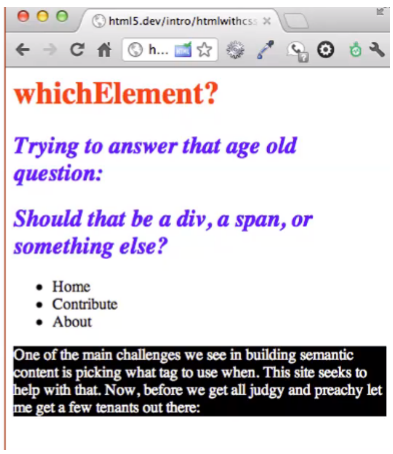
11.2 HTML

HyperText Markup Language è un linguaggio di markup per **dare struttura ai contenuti** del web. La struttura data al contenuto viene esplicitata tramite i **tag**. Essi *delimitano* dei pezzi di documento e possono essere composti da altri tag. Ogni tag ha un suo particolare effetto.
In assenza di CSS, il browser si "arrangia" nel dare una rappresentazione grafica standard del documento.

11.3 CSS

Cascading Style Sheets è un linguaggio di markup per **definire uno stile** ai contenuti del web.

```
<style type="text/css">
    h1{
        color: red;
    }
    h2{
        color: blue;
        font-style: italic;
    }
    p{
        color: white;
        background-color: black;
    }
</style>
```



11.3.1 Selectors

Possiamo specificare un selettore più preciso rispetto al nome del tag.

```
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

11.3.2 Media Query

Le media query possono essere viste come particolari selettori capaci di **valutare le capacità del device** di accesso alla pagina.
Servono per controllare ad esempio:

- Larghezza e Altezza del device
- Orientamento dello schermo
- Risoluzione

Oggi come oggi, gli stili dei siti web sono *responsive*, ovvero che lo stile di rappresentazione della pagina varia in base alla larghezza dello schermo. (Quindi in base al device di accesso alla pagina).

11.3.3 Perchè cascading?

Esistono potenzialmente diversi stylesheets:

- L'autore della pagina in genere ne specifica uno o pià di uno
- Il browser ne ha uno, o un vero e proprio CSS o simulato nel loro codice
- Il lettore, l'utente del browser ne può definire uno proprio per customizzare la propria esperienza

Sono quindi inevitabili dei **conflitti** ed è necessario definire un algoritmo per decidere quale stile vada applicato ad un elemento.

11.4 DOM

Document Object Model è una **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 del web.
Ogni nodo può essere caratterizzato da attributi che ne facilitano l'identificazione, la ricerca, la selezione:

- **Identificatore Univoco**
- **Classe** che indica l'appartenenza ad un insieme che ci è utile definire