

Abstract

Questo è un documento in cui si collezionano gli appunti sulla parte 1 del corso di *Reti di Calcolatori e Principi di Cybersecurity*. Quindi a partire dall'introduzione alle reti fino alla trattazione esaustiva dei protocolli DNS, HTTP, SMTP, POP e i suoi aspetti collegati.

Si sconsiglia quindi di usare questo materiale *in alternativa* alle lezioni o alle slide fornite dal docente A. Bartoli, in quanto i miei appunti potrebbero contenere refusi, errori di battitura, oppure potrebbero essere difficili da comprendere senza aver avuto un'idea dei contenuti a priori, che sono ottenibili seguendo le lezioni.

Detto ciò, non mi assumo nessuna responsabilità del vostro rendimento in questo corso se decidete di basarvi su questi appunti.

Dino Meng

"Introduzione alle Reti di Calcolatori"

Processi Client e Server

X

*Fondamenti sulla comunicazione tra processi. Paradigma processo client e processo server.
Osservazione sulla terminologia "server".*

X

0. Voci correlate

- Concetto di Processo

1. Processi Client e Server

Richiamiamo dal corso di Sistemi Operativi che un *processo* è un programma in esecuzione. Come ben sappiamo, più processi possono essere *comunicanti tra di loro*: nel tema della sincronizzazione, si sofferma su *come* possiamo coordinare i processi che stanno eseguendo sullo stesso calcolatore.

Tuttavia, notiamo che i processi possono essere anche *distribuiti* su calcolatori diversi e comunque comunicarsi! In un certo senso, possiamo avere dei calcolatori "*localizzati potenzialmente ovunque*".

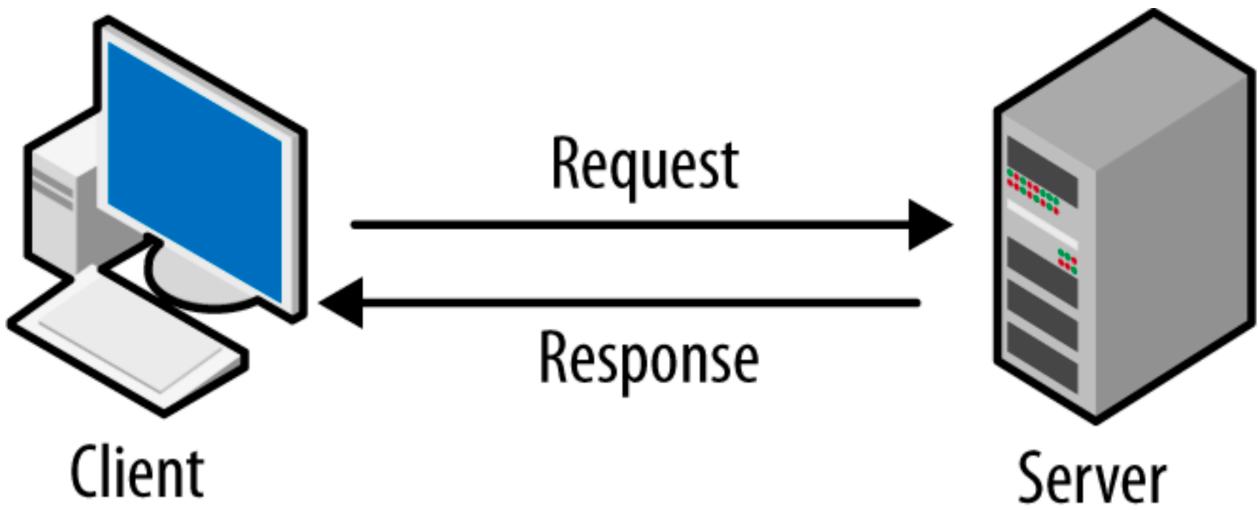
Il paradigma principale per la comunicazione tra processi è il modello *client-server*, in cui diamo le seguenti definizioni.

- *Processo Server*: Offre un "*servizio*" ad altri processi
- *Processo Client*: Usa un "*servizio*" offerto da altri processi

Vedremo cosa si intende per "*servizio*" nelle parti successive del corso.

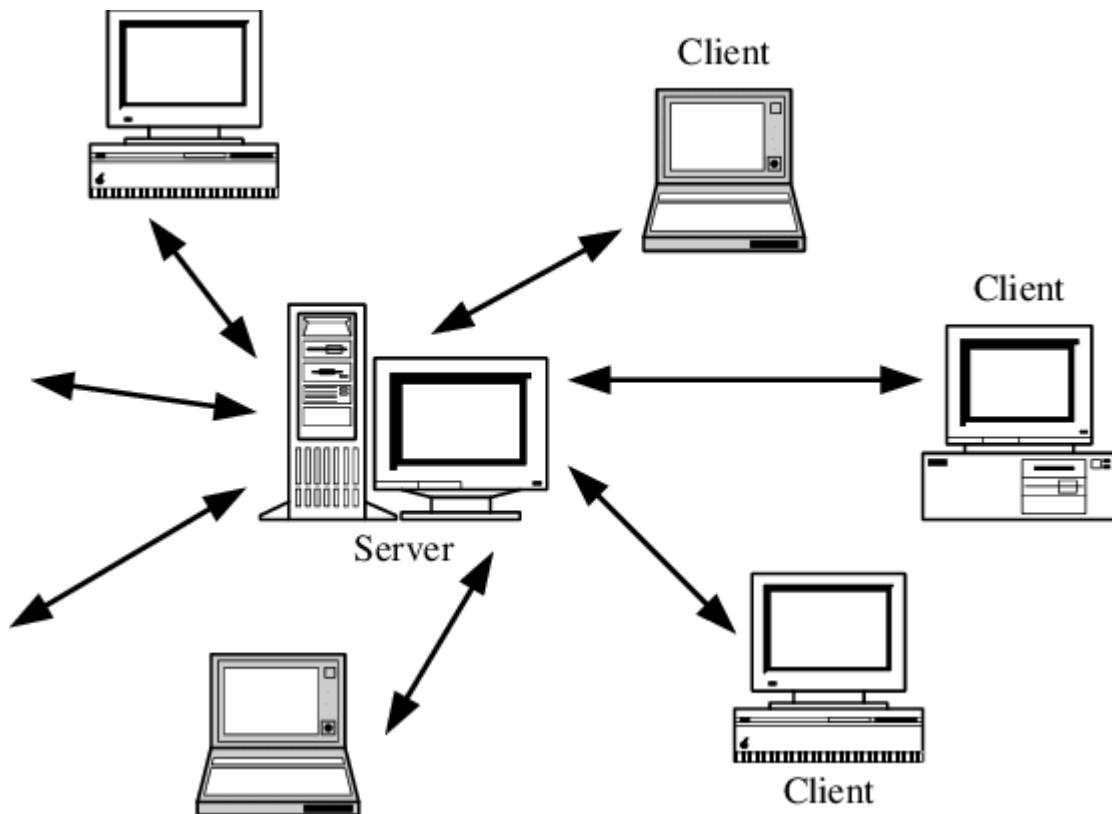
Nel modello, si esegue il seguente schema (ultra semplificato!)

1. *Client* invia una *richiesta* al *Server*
2. Il *Server* riceve la richiesta, la interpreta ed esegue il servizio
3. Il *Server* trasmette una risposta al *Client*



Stiamo effettuando una semplificazione del vero modello in quanto possiamo avere a che fare con casistiche più flessibili! In stesso intervallo temporale possiamo verificare:

- **Concorrenza:** Un *client* può usare servizi di più *server* diversi o viceversa (un *server* offre più *servizi* a *client diversi*)
- **Simultaneità:** Un processo può essere sia che *server* che *client*



Esempio. Per fare un esempio, prendiamo il *web* (semplificando la realtà):

- Un *Processo Client* sarebbe un browser in esecuzione
- Un *Processo Server* sarebbe il web server. Può prestare molti servizi, come prelevare documenti, eseguire programmi o effettuare delle query sul database, tutto fatto nel server.

2. Terminologia Server (!)

Prestiamo attenzione che *server* può riferirsi a due significati:

- Il *processo*, come faremo in questo corso
- Il *calcolatore* la cui funzione principale è *eseguire processi server sempre attivi*, come si intende di solito nel linguaggio comune

Comunicazione tra Processi

X

Comunicazione tra processi. Livello fisico: definizione di internet. Livello logico: definizione di connessione. Struttura di un client e di un server.

X

0. Voci correlate

- Concetto di Processo
- Sistema Client-Server

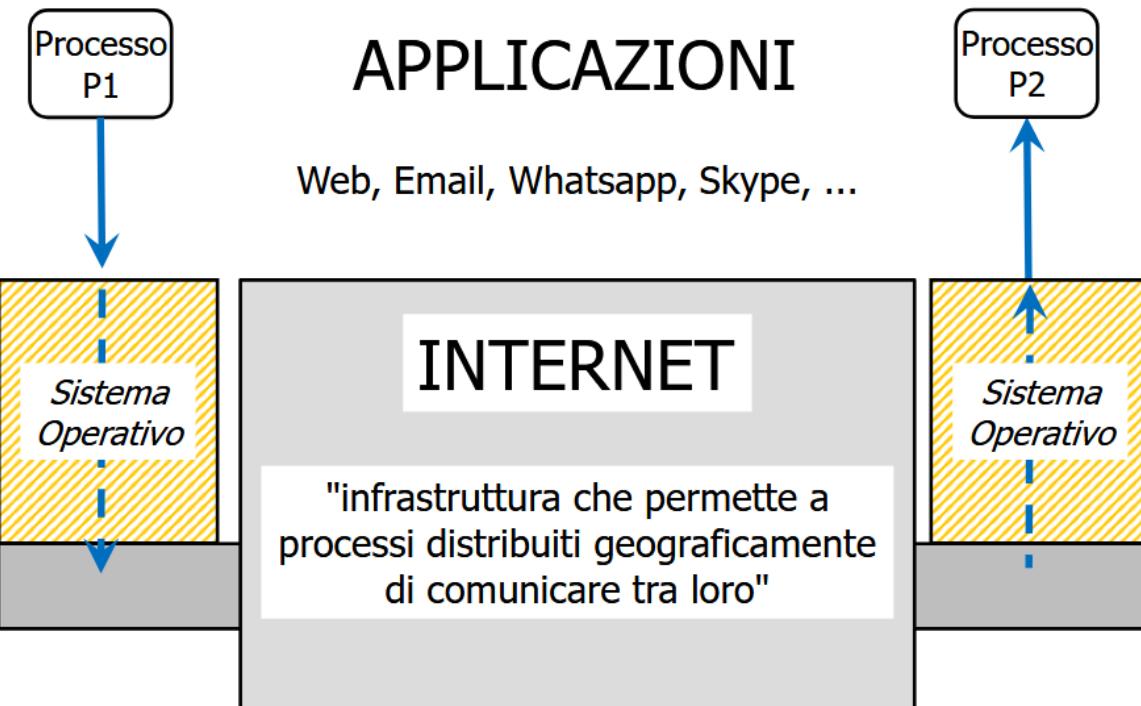
1. Comunicazione tra Processi

1.1. Livello Fisico

Supponiamo di avere due processi, P_1 e P_2 , ove P_1 desidera mandare un messaggio m al processo P_2 . Per farlo, P_1 effettuerà delle *System Call* (Libreria Standard e System Call) per istruire al sistema operativo di mandare il messaggio mediante mezzo fisico.

Dopodiché, si localizza P_2 e mediante l'*internet* il messaggio viene comunicato al calcolatore in P_2 e poi tramite delle system call riceve il messaggio.

Il passaggio cruciale (in questo caso) è proprio l'*internet*, che è una *infrastruttura che permette tale processo di comunicazione*.



1.2. Livello Logico

Ci focalizziamo sul *"come si usa"* questo sistema, senza soffermarci sui tecnicismi fisici.

Osserviamo che mediante i sistemi operativi, si va a creare una *connessione logica* in cui i processi sono in grado di comunicarsi tra di loro, inviando dei semplici comandi al proprio sistema operativo. Essa funziona come un *"tubo"*, in cui il client è in grado di inviare richieste e ricevere risposte. In un certo senso, è un'illusione prodotto dal sistema operativo.



Vediamo come i processi comunicanti useranno questo "*tubo*" (connessione logica)

Client:

1. Determina l'identificatore del server (vedremo nella parte successiva del corso)
2. Apre la connessione verso l'identificatore del server
3. Finché non si decide di chiudere la connessione,
 1. Trasmettere una *Request*
 2. Ricevere una *Response*

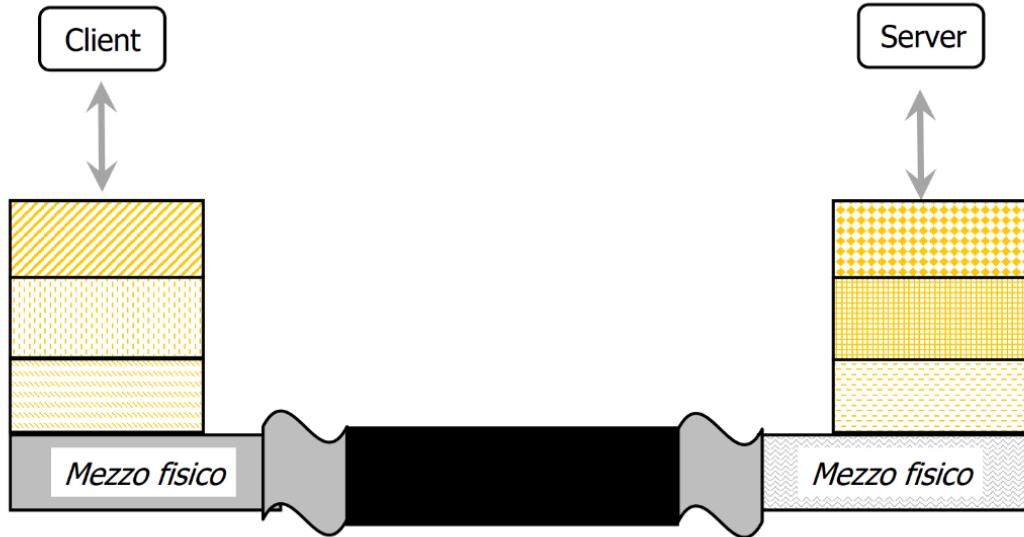
Server:

1. Sceglie il proprio identificatore
2. Attendere richieste di apertura verso il proprio id, quindi è in "*standby*"
3. Ogni volta che mantiene la connessione, finché non si chiude:
 1. Ricevere una *Request*
 2. Elaborare la *Request* internamente
 3. Trasmettere una *Response*

X

3. Rete Software

Notiamo che i *sistemi operativi* e i *mezzi fisici* associati ai processi possono essere *diversi*. In particolare, il *software di rete* è internamente suddiviso in *3 layer* (vedremo dopo perché).



Ogni layer dovrà implementare lo *stesso layer del protocollo corrispondente*, per "capirsi tra di loro" (anche se non è vero ma supponiamo vera questa cosa, con uno sforzo di fantasia)

I layer sono le seguenti:

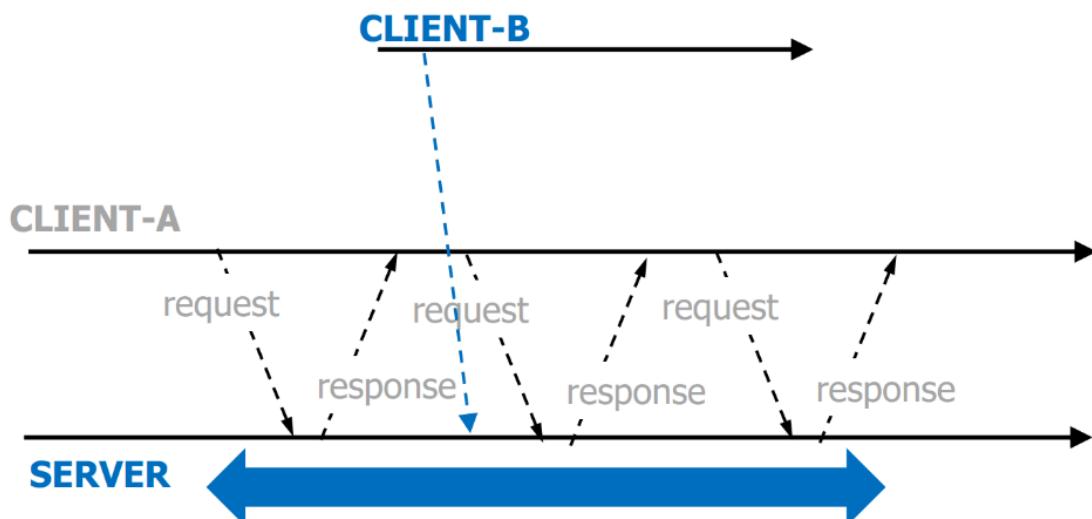
- *Protocollo IP*: Livello Intermedio
- *Protocollo TCP o UDP*: Livello più alto, e verranno "usate" dai processi.

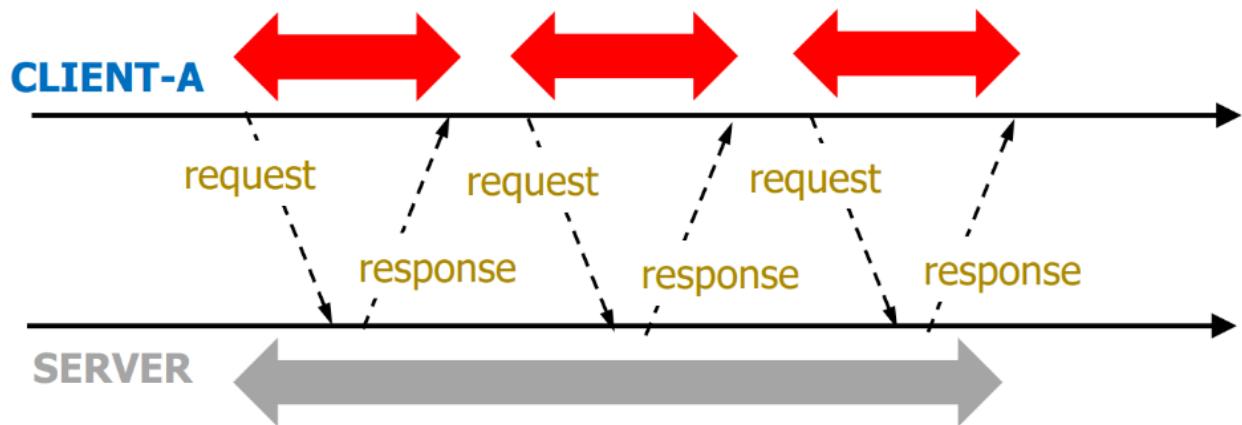
Nel corso vedremo i protocolli che usano *TCP*. L'unica l'eccezione sarà il protocollo *DNS*, che può funzionare sia su *UDP* che *TCP*.

 X

4. Concorrenza

Q. Nel modello *client-server*, è possibile che più *client* comunichino con lo stesso server (e viceversa). Si vuole gestire la *concorrenza*? Ovvero, ha senso che il *server* mantenga un client in attesa fino a quando l'altro client ha chiuso la connessione? Oppure, se è ragionevole che un client rimanga inattivo tra invio request e ricezione risposte?





Per dare una risposta, osserviamo le scale temporali. Riportiamo le seguenti *scali temporali* per un calcolatore:

- 1 Ciclo CPU: 0.3 ns
- Accesso alla DRAM dalla CPU: 120 ns
- Ping: Un numero variabile, da 40ms a 183ms

Se normalizziamo questa scala rispetto ad un ciclo CPU (quindi assumendo che duri un secondo), si avrebbe

- 1 Ciclo CPU: 1 s
- Accesso alla DRAM dalla CPU: 6 min
- Ping: Un numero variabile, da 4 anni a 19 anni (!)

Quindi confrontando le *scale* dei tempi, vediamo che è necessario implementare *server* e *client* che siano in grado di gestire comunicazioni parallele e concorrenti.

Indirizzo TCP

X

Metodo di identificazione dei processi nelle reti. Indirizzo TCP: definizione, indirizzo IP e port number. Notazione IP: dotted decimal notation. Relazione tra indirizzo IP e la relativa posizione geografica del calcolatore. Scelta dell'indirizzo TCP dei processi server. Assegnazione dell'indirizzo TCP dei processi client.

X

0. Voci correlate

- Comunicazione tra Processi

1. Indirizzo TCP (semplificato)

Q. Supponiamo di avere due processi che vogliono comunicare tra di loro. Come vengono *identificati* i processi sull'internet? In particolare, come garantiamo che la maniera con cui vengono identificati siano *univoci*?

Una soluzione è fornita dall'*indirizzo TCP*.

DEFINIZIONE. Ogni *processo* collegato a Internet è identificato univocamente dal *suo indirizzo TCP*, una coppia formata dall'*indirizzo IP* e il *port number*.

- L'indirizzo IP è un numero naturale compreso nell'intervallo $[0, 2^{32} - 1] \approx [0, 4 \cdot 10^9]$ e identifica il *nodo* (inteso come *calcolatore* sull'internet)
- Il port number è un numero naturale compreso nell'intervallo $[0, 2^{16} - 1] = [0, 65535]$ e identifica il *processo* sul nodo

L'indirizzo IP è *assegnato* al nodo, il port number invece è "*scelto*" dai processi server e *assegnato* ai processi *client*.

X

2. Proprietà dell'Indirizzo IP

2.1. Notazione Indirizzo IP (IPv4)

Un modo comune per rappresentare un indirizzo IP è la *dotted decimal notation*. Essa consiste in prendere la rappresentazione binaria dell'indirizzo IP (quindi ho 32 bit), suddividerlo in 4 byte e poi di rappresentare ogni byte in decimale; infine li mettiamo assieme, separandoli con un punto.

Esempio: **11000000|10101000|00000001|00000001** diventa **192.168.1.1**

2.2. Relazione (approssimativa) con la Posizione Fisica

Premettiamo che l'indirizzo IP non è un *localizzatore*, in quanto essa serve principalmente ad *identificare*. Tuttavia, vedremo che in certi casi è possibile "*dedurre*" delle informazioni sulla posizione geografica

- L'indirizzo IP rappresenta una posizione geografica *molto approssimata*, di solito con grande errore. Ad esempio, se l'indirizzo IP ci punta al campus principale dell'Università degli Studi di Trieste, non sarà possibile sapere in quale aula esatta si trova il nodo
- Dei "*nodi molto vicini*" possono avere indirizzi IP "*molto simili*", soprattutto nei byte a destra
- Ogni volta che un nodo si sposta a distanza sufficiente, deve cambiare l'indirizzo IP

Approfondiremo questo aspetto nella parte 2 del corso

X

3. Determinazione dell'Indirizzo TCP

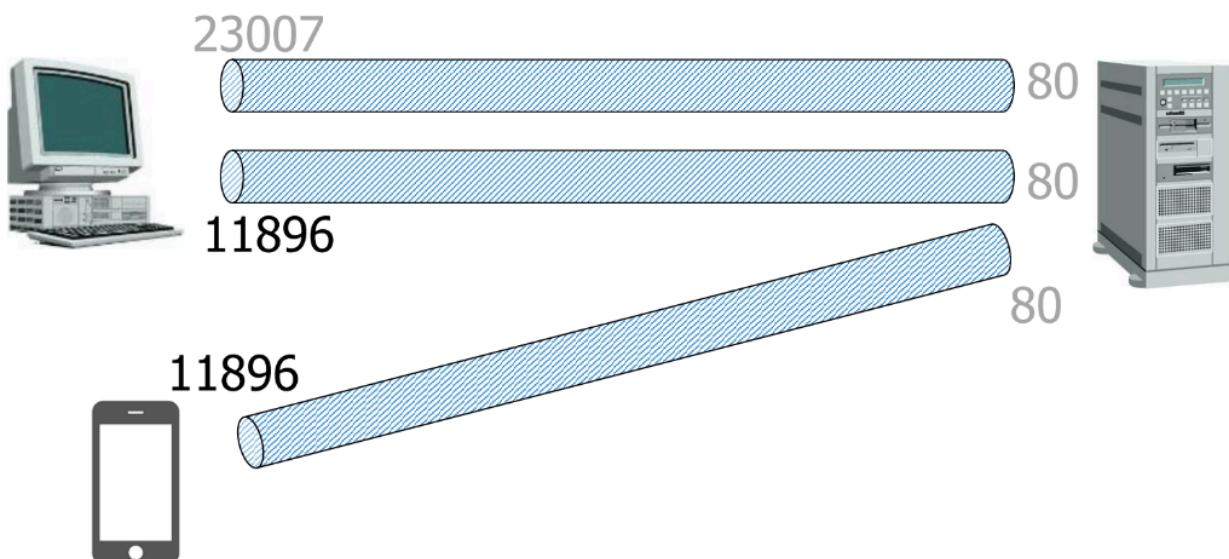
Client:

- Il port number viene scelto dal *sistema operativo* e *sul momento*. Esso è diverso da ogni altro port number già in uso, e dev'essere superiore a 1024 (per motivi storici, si vuole evitare di occupare i port che userebbero un server)
- Può (e di solito succede) cambiare ad ogni esecuzione

Server:

- Si sceglie il port number in base al *protocollo* da implementare e viene fissato

Osserviamo che, dato un server, se due processi client su nodi diversi si collegano al server allora *possono* avere lo stesso port (in quanto gli IP saranno sicuramente diversi)



Ricerca di Indirizzi IP

X

Ricerca di indirizzi IP per client. Esempi introduttivi: Protocollo HTTP, protocollo SMT/POP. Schema generale.

X

0. Voci correlate

- Indirizzo TCP
- Comunicazione tra Processi

1. Esempi Introduttivi di Ricerca Indirizzo IP

Q. Supponiamo che un client voglia collegarsi ad un server. Come primo passo, deve identificare l'indirizzo IP del server. Come fa?

Adesso vediamo con un paio di esempi.

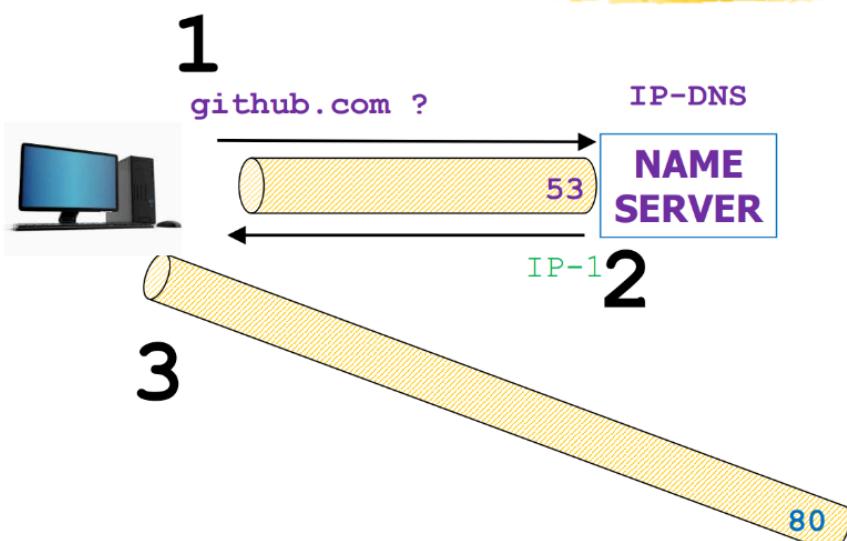
1.1. HTTP

Step 1: Conoscendo l'**URL** della *pagina web*, traiamo il *nome server*

- *Esempio:* **https://www.units.it/catalogo-della-didattica-a-distanza** è l'URL, otteniamo il nome server **www.units.it**

Step 2: Conoscendo il *nome server*, ottenerne l'**IP-server**. Per farlo, bisogna interagire col *protocollo DNS*. Essa consiste in inviare una *richiesta* del tipo "**indirizzo IP di <nome_server>?**" e ricevere la risposta contenente l'**IP-server**, da cui concludiamo.

- Daremo per scontato l'**IP-DNS** e la sua *port number* conosciuti. In particolare il port sarà 53.



1.2. Email

Step 1: Conoscendo il *proprio indirizzo email* (conosciuto a priori, diciamo che è fa parte della *configurazione*), ottenere il nome server

- *Esempio:* **bartoli.alberto@units.it** diventa **mx.units.it**
- Vedremo come funziona nei dettagli dopo

Step 2: Come prima, interagire col server DNS per ottenere l'*IP-Server*

X

2. Schema Generale

Step 1. Dipendente dal protocollo e dalla configurazione

Step 2. Dal *nome server* otteniamo l'*IP-Server*, basandoci sulla DNS (quindi ci servirà l'*IP-DNS* in configurazione)

Osserviamo che la *step 2* è la parte cruciale dello schema di ricerca di indirizzi IP.

Socket

X

Socket per implementare IPC ad alto livello. Definizione di socket, interfaccia TCP semplificata (primitive). Struttura di un processo server (idea e pseudocodice). Struttura di un processo client (idea e pseudocodice).

X

0. Voci correlate

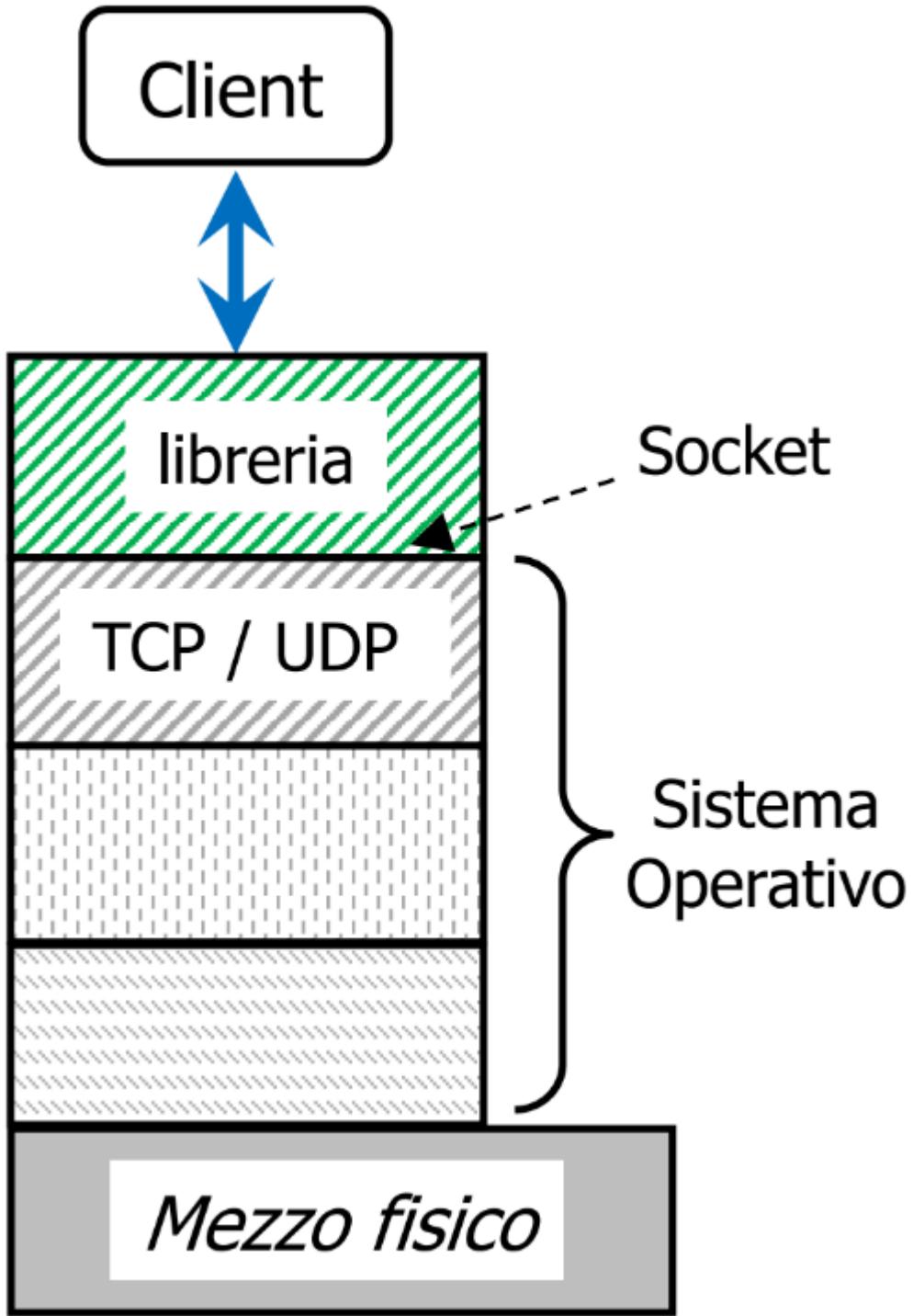
- Comunicazione tra Processi
- Socket

1. Socket

Dato un *collegamento* tra due *processi*, abbiamo che i sistemi operativi realizzano questo "*tubo*" che li permette di inserire e trasmettere messaggi. Vediamo nei dettagli come funzionano questi meccanismi, in particolare le *System Call*.

Chiamiamo i *socket* le *interfacce di programmazione* (insieme di funzioni/primitive) attraverso le quali si accedono *TCP* o *UDP*. Sono considerate *quasi standard* e molto complicate.

Molto spesso delle *librerie* (e.g. Python) implementano l'interfaccia socket con un livello di astrazione ancora più alto, tuttavia non sono più considerati standard (e dipende dal linguaggio).



2. Interfaccia TCP

Vedremo come, in una maniera semplificata, i socket implementino l'interfaccia **TCP**. Si hanno le seguenti system call (primitive):

- **SOCKET**: Va a creare l'estremità del collegamento
- **SEND**: Mandare dati nella connessione
- **RECEIVE**: Ricevere dati dalla connessione
- **CLOSE**: Chiudere la connessione

Queste primitive vengono utilizzate sia da processi client che server. Vediamo adesso le primitive specifiche necessarie per implementare la struttura di un client e di un server.

2.1. Server

Ricordiamo che il server:

Server:

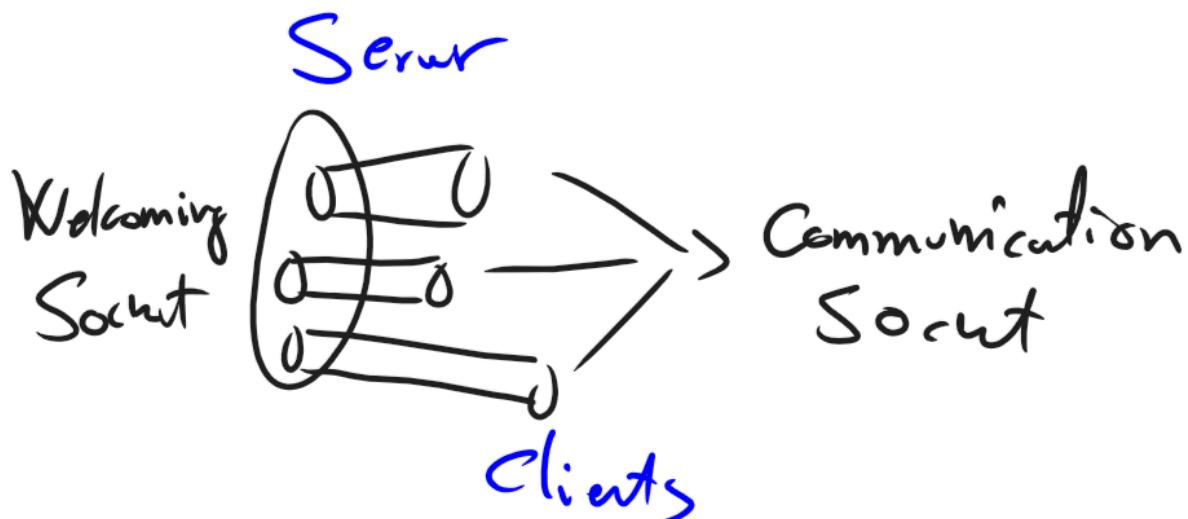
1. Sceglie il proprio identificatore
2. Attendere richieste di apertura verso il proprio id, quindi è in "*standby*"
3. Ogni volta che mantiene la connessione, finché non si chiude:
 1. Ricevere una *Request*
 2. Elaborare la *Request* internamente
 3. Trasmettere una *Response*

Adesso vediamo come implementiamo gli step 2 e 3.

PSEUDOCODICE.

- Creare socket s_1 (*SOCKET*)
- Scegliere un port number (da fare seguendo i protocolli)
- Associare s_1 al port number scelto (*BIND*)
- Dichiara disponibilità a connessioni su s_1 (*LISTEN*)
- Attendere connessione su s_1 . Non appena si ottiene una connessione, accettare (*ACCEPT*):
 - Ottenere un altro socket s_2 e comunicare su s_2 col client

Notiamo che in effetti ci sono più server socket: abbiamo un *welcoming socket* che attende le connessioni, e più *communication socket* per ricevere dati



Osservazione:

- Il server non ha bisogno di conoscere l'indirizzo TCP dei client, anche se può conoscerli con determinate funzioni in casi particolari

- Il server usa il proprio port number *soltamente una volta*, ovvero nel *bind* col welcoming socket
- Il server non conosce i port number dei *communication socket*, e non ha necessità di conoscerli. Infatti, hanno lo stesso numero di porta; vedremo che il sistema operativo è in grado di distinguerli, ma non in questo corso

2.2. Client

Ricordiamo la struttura di un client:

Client:

1. Determina l'identificatore del server (vedremo nella parte successiva del corso)
2. Apre la connessione verso l'identificatore del server
3. Finchè non si decide di chiudere la connessione,
 1. Trasmettere una *Request*
 2. Ricevere una *Response*

PSEUDOCODICE.

1. Crea socket *s* (*SOCKET*)
2. Connnette *s* con (IP-SRV, PORT-SRV) (*CONNECT*)
3. Comunica su *s* in base al protocollo usato

Osservazioni:

- Il client non si accorge che sul lato server ci sono *due socket*
- Il client non conosce il proprio port number, e non avrà necessità di conoscerlo in nessun caso (può farlo lo stesso)

Protocolli di Testo e Binari

X

Breve descrizione qui

X

0. Voci correlate

- Comunicazione tra Processi

1. Definizione di Protocollo

Abbiamo visto che abbiamo una *collegamento* tra due processi è un "*tubo*" in cui possiamo trasmettere dei dati. Ma come sono rappresentati questi dati?

Vengono trasmessi sotto forma di *byte*, ovvero un numero naturale in $[0, 255] \cap \mathbb{N}$. Questo ha conseguenze importanti! Infatti, enunciamo il *principio di openness*

Openness: I client e server possono essere sviluppati da organizzazioni diverse e in luoghi e tempi diversi.

Dal principio sorge il problema di *interpretare* effettivamente i dati, ovvero come i processi possano "*capire*" tra di loro.

Protocolli: La soluzione immediata e la più semplice è quello di usare i *protocolli*, quindi un insieme delle *regole note a priori*. In particolare possiamo avere le seguenti tipologie di regola:

- Sintassi (*struttura*)
- Semantica (*significato*)
- Tempistiche

I processi che partecipano in una applicazione devono implementare il *protocollo di quell'applicazione*.

Di solito vengono gestiti apertamente dalla *IETF* (Internet Engineering Task Force) e sono descritti nei documenti chiamati *Request for Commenters* (RFC). Ciò rende libero lo sviluppo di processi client e/o server.

Tuttavia, ci sono casistiche rari in cui i protocolli vengono sviluppati privatamente e sono mantenuti riservati e/o coperti da copyright (ex: Skype).

X

2. Categorie di Protocolli

Partiamo dal presupposto che *ogni byte* trasmesso viene rappresentato in sistema *esadecimale*.

Ci sono principalmente *due categorie di protocolli*: *testo* e *binari*.

Testo:

- *Ogni byte viene rappresentato sotto forma di un carattere.* Solitamente si usa la convenzione ASCII. Per codificare la *linea di testo*, si usa il *line feed* e *carriage return* (codificate da **0A** e **0D**)
- Esempi: HTTP, SMT POP

Binario:

- Ogni *bit* ha un significato specifico e non può essere tradotto in testo. Alcuni byte hanno valori che corrispondono a caratteri ASCII, ma il messaggio rimane comunque non interpretabile. I *software* come *Wireshark* permettono di dare una interpretazione.
- Esempio: DNS

Proprietà dei Servizi di Comunicazione

X

Proprietà dei servizi di comunicazione. Definizione di servizio connection-oriented o connectionless; byte-oriented o message-oriented; reliable o unreliable. Combinazioni tipiche: TCP, UDP.

X

0. Voci correlate

- Comunicazione tra Processi

1. Proprietà Servizi di Comunicazione

Dato un *servizio di comunicazione*, lo immaginiamo come un intermediario che manda messaggi. Nell'esempio fisico, si avrebbero le poste. Quindi non tutto è garantito, tra cui che *arrivino* effettivamente i messaggi, oppure che *arrivino in ordine*. Quali garanzie possiamo offrire a chi *usa* il servizio di comunicazione?

1.1. Orientamento con la Connessione (collegamento)

DEFINIZIONE. Un servizio di comunicazione si dice *communication oriented* (o *connection*) se essa segue questo schema:

- Si apre la connessione
- Si *trasmette o riceve* nella connessione
- Si chiude la connessione
(in parole semplici, va ad usare un "*tubo*" da aprire prima delle trasmissioni)

Quindi è necessaria un'apertura *esplicita* prima dell'utilizzo, in cui specifico il destinatario. Notiamo che implicitamente essa garantisce l'*ordinamento* dei messaggi, i.e. l'*ordine di ricezione* è l'*ordine di trasmissione*.

DEFINIZIONE. Un servizio di comunicazione è invece *connectionless* se invece non richiede un'apertura preventiva di nessuna connessione. Ogni *trasmissione* indica il destinatario e trasmittente; in questo caso, non manteniamo l'*ordinamento* in quanto ogni trasmissione è indipendente dall'altra.

1.2. Orientamento su Messaggi o Byte

DEFINIZIONE. Un servizio di comunicazione è:

- *Byte-oriented* se delle trasmissioni consecutive creano un *unico flusso di byte*, quindi il destinatario può ricevere più trasmissioni allo stesso tempo

- *Message-oriented* se ogni trasmissione è un messaggio separato dagli altri

Osserviamo quindi che in un servizio *byte oriented*, si ha che i byte mandati e ricevuti *sono diversi!*

Inoltre, in un protocollo *byte-oriented* potrebbe essere anche necessario avere delle *regole* per partizionare il flusso di byte in messaggi.

1.3. Reliability

DEFINIZIONE. Un servizio di comunicazione si dice *reliable* se tutti i dati sono ricevuti e ogni dato viene ricevuto *solo una volta*. Invece è *unreliable* se non è *reliable*, quindi o i dati possono essere persi o i dati possono essere ricevuti più volte.

N.B. In realtà la nozione di *unreliability* è *molto più complessa*, e va a tenere altri fattori. In questo corso ci faremo andare bene con questa definizione.

2. Combinazioni tipiche delle Proprietà

TCP:

- Communication-oriented
- Reliable
- *Byte-oriented* (!)

UDP:

- Connectionless
- Unreliable
- Message-oriented

Naturalmente, *TCP* è sempre l'alternativa migliore (anche se leggermente più costosa, da un punto di vista computazionale). L'*UDP* è esistito per motivi storici.

OSSERVAZIONE. Nessuna delle proprietà danno una garanzia di ordine temporale: per i servizi di comunicazione, il tempo "*non esiste*".

"DNS"

Prime Nozioni sull'Internet

X

Prime nozioni sull'internet, per il DNS. L'internet a grandi linee: dispositivi che si collegano ai router. Definizione di organizzazione, router di frontiera. Esempi di organizzazioni. Definizione di Host.

X

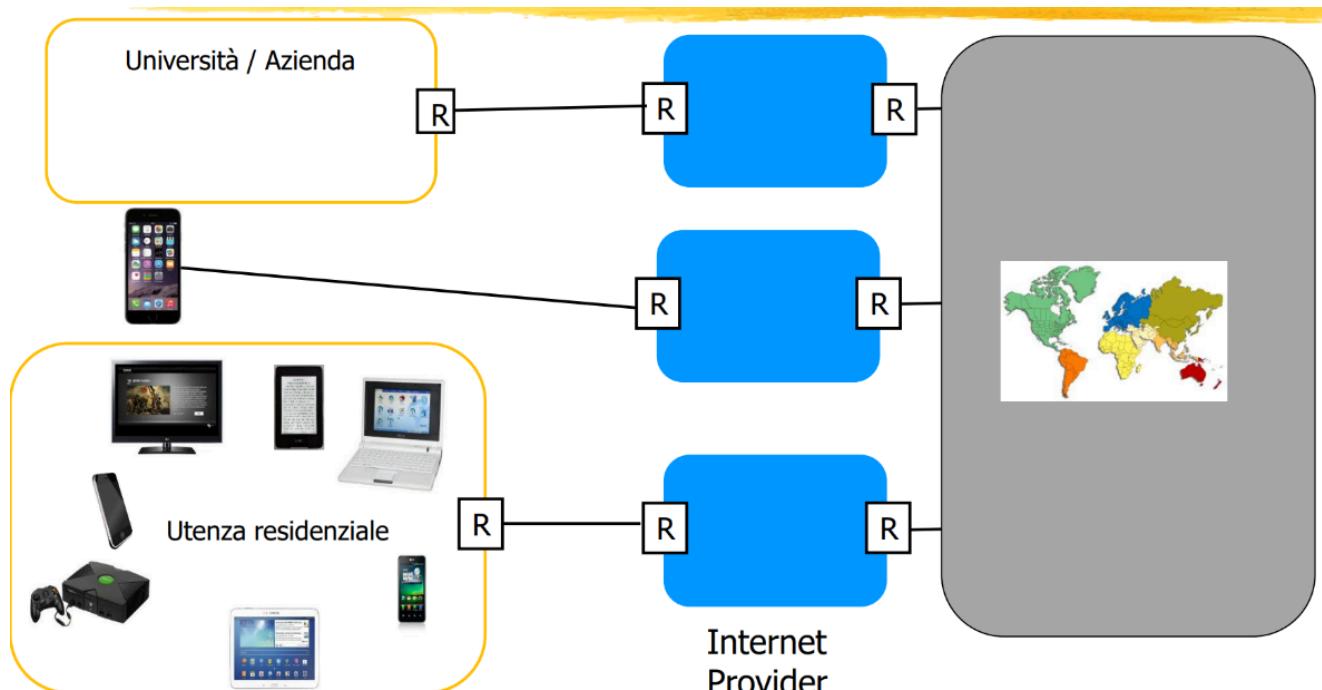
0. Voci correlate

- Comunicazione tra Processi

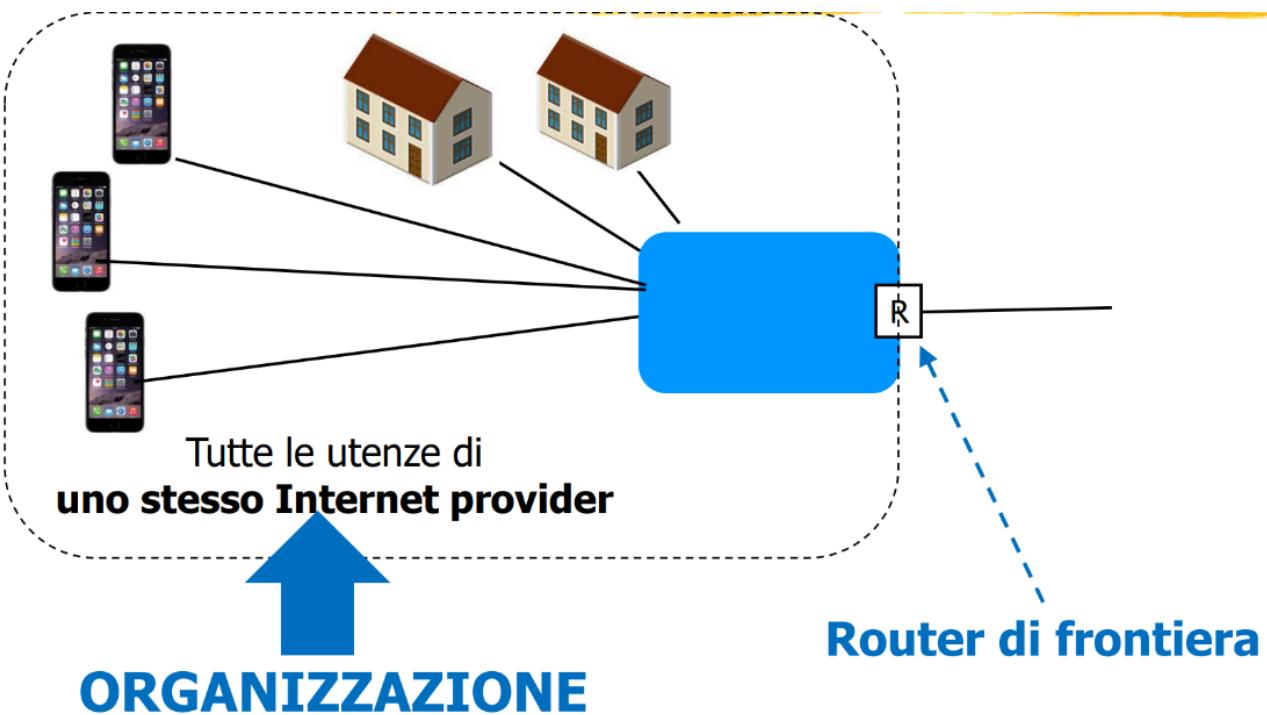
1. Prime Nozioni sull'Internet

A grosso modo, era stato definito l'*internet* come un'infrastruttura fisica che permette ai calcolatori di comunicare tra di loro. Adesso, diamo un paio di cenni di com'è strutturato l'internet.

A ogni dispositivo si collega ad un *router di frontiera*, che è un calcolatore da cui transita tutto il traffico all'internet. I router di frontiera sono gestiti dagli *internet provider*, organizzazione di mestiere.



Diciamo un'*organizzazione* tutte le utenze di uno stesso *Internet provider*, quindi che sono collegati allo stesso router di frontiera.



Esempi:

- PC collegato ad Internet da casa: appartiene al provider a cui si è iscritto, come Fastweb
- Lo stesso PC collegato ad Internet con eduroam: adesso appartiene all'Università a cui è collegata

Q. Come mai ogni dispositivo deve collegarsi ad un'organizzazione?

Nella pratica, i *router di frontiera* possono essere configurati per bloccare certi tipi di traffico (analisi sui numeri di porta) e anche alcuni indirizzi IP (vedremo con DNS).

DEFINIZIONE. Diremo *host* un calcolatore collegato a Internet. Il termine "*nodo*" può creare ambiguità, in particolare nel conteso delle *DNS*. Ogni host ha un *indirizzo IP* e può avere un nome.

X

DNS: Motivazioni, cos'è. Come riesce ad effettuare il Name Resolution. Osservazioni sul DNS.

X

0. Voci correlate

- Prime Nozioni sull'Internet
- Ricerca di Indirizzi IP
- Comunicazione tra Processi

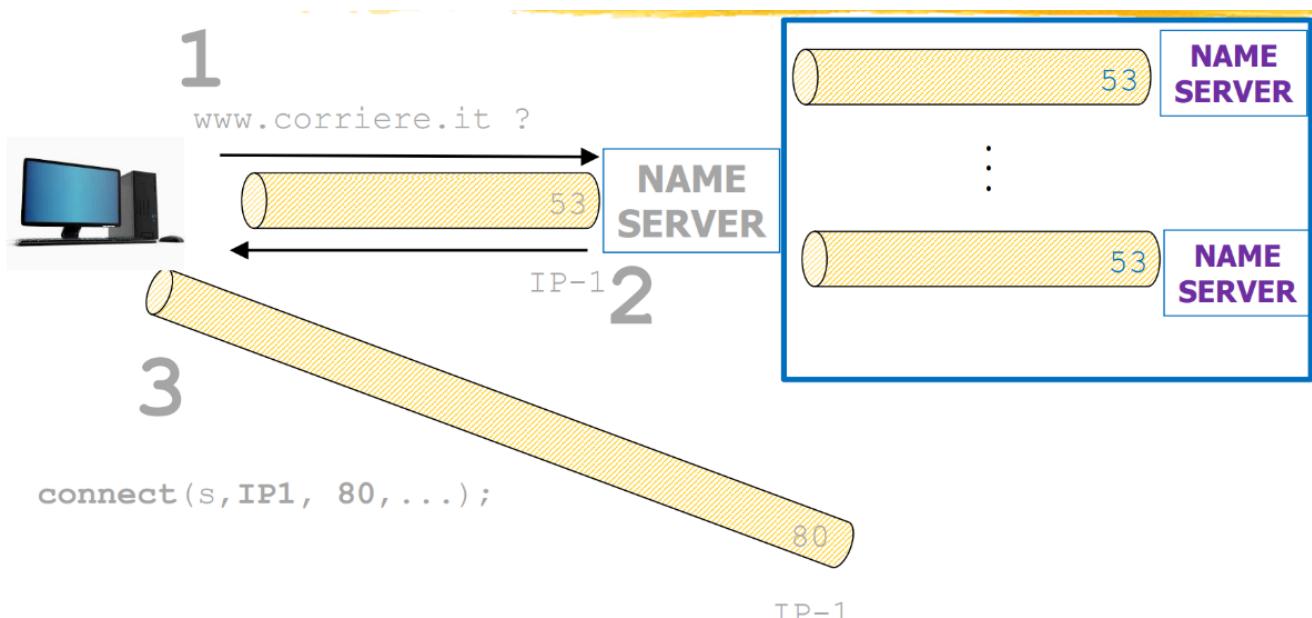
1. Nomi degli Host

Ogni *host* ha un *indirizzo ip* e può avere un *nome*. Il nome è utile per *aiutare* a individuare gli host, in quanto individuarli tramite indirizzo IP (anche indispensabili) è *scomodo*.

DOMAIN NAME SERVER. Comunque, per collegarsi ad un altro calcolatore, è comunque necessario risalire al suo indirizzo IP. I *Domain Name Server* (DNS) servono principalmente per realizzare una *tavella di traduzione* tra *name* e i loro corrispondenti *value* (indirizzo IP).

- I *DNS* sono centinaia di migliaia di name server distribuiti *geograficamente*
- Ognuno è "circa" gestito da un'organizzazione diversa
- Tuttavia ogni client avrà l'illusione di vedere un solo *DNS* (server DNS)

DEFAULT NAME SERVER. Ricordiamo il processo per *trovare* l'indirizzo IP di un host, bisogna avere l'indirizzo IP di un "*server DNS*" (Ricerca di Indirizzi IP). Questo *server DNS* andrà in realtà (tipicamente) a contattare *altri name server*.



Non vedremo nei dettagli *come* si effettuano le altre chiamate su DNS. Facciamo tuttavia un paio di osservazioni più utili a fine pratici:

- L'indirizzo IP del *Default Name Server* dev'essere noto a priori!
- Ogni *Name Server* opera sia da *server* che *client*, fornendo un esempio di simultaneità
- Ogni *host* contatta *sempre e solo* il proprio name server. La ricerca effettiva viene fatta *solo* dal *Default Name Server*, non gli altri *DNS* (intesi come *Domain NS*).
- Nel DNS i nomi terminano sempre col carattere `.`, da tenere conto per gli esercizi

Notiamo infine che *DNS* è un termine ambiguo, in quanto può riferirsi:

- Alla *intera infrastruttura mondiale* che realizza la tabella di ricerca *name-value*
- Ad un *name server specifico*, i.e. il *Default Name Server* o un'altro
- Al *protocollo di comunicazione* usato per interagire coi name server

Architettura DNS

X

Cenni di descrizione fisica dei DNS. Modi di configurare i DNS degli host, a seconda delle organizzazioni. DNS Centralizzati, DNS non centralizzati.

X

0. Voci correlate

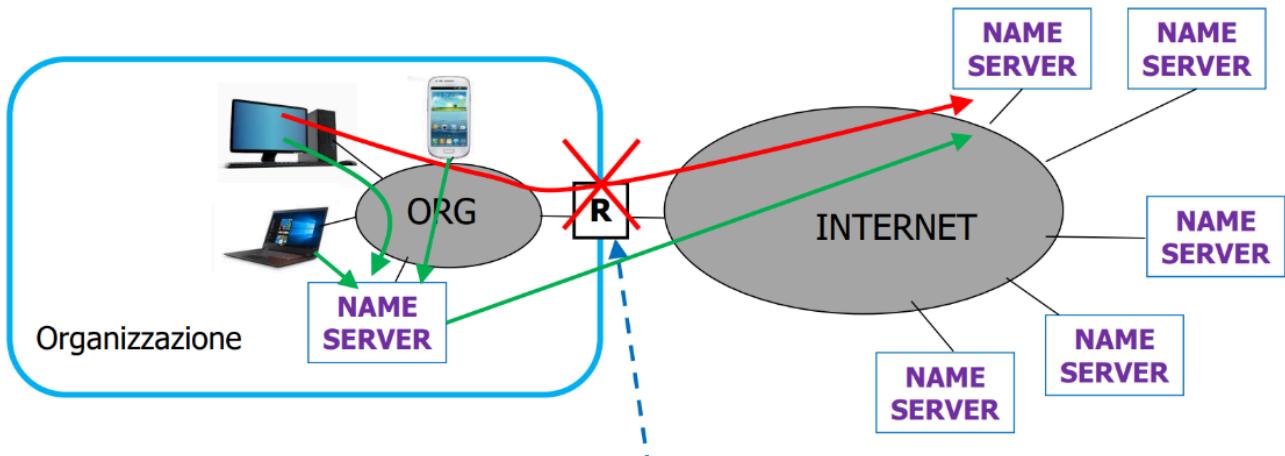
- Prime Nozioni sull'Internet
- Name Server

1. Architettura Name Server

Sappiamo che ogni host è *configurato* con *l'indirizzo IP* del proprio *DNS*. Ci sono due casi in cui configuriamo il *DNS*:

Caso 1 (tipico "aziende/università", centralizzati)

- Il DNS è "*interno ad organizzazione*" e *tutti gli host* appartenenti dell'organizzazione hanno quel *DNS*
- Si permette *solo* il traffico DNS a quel name server

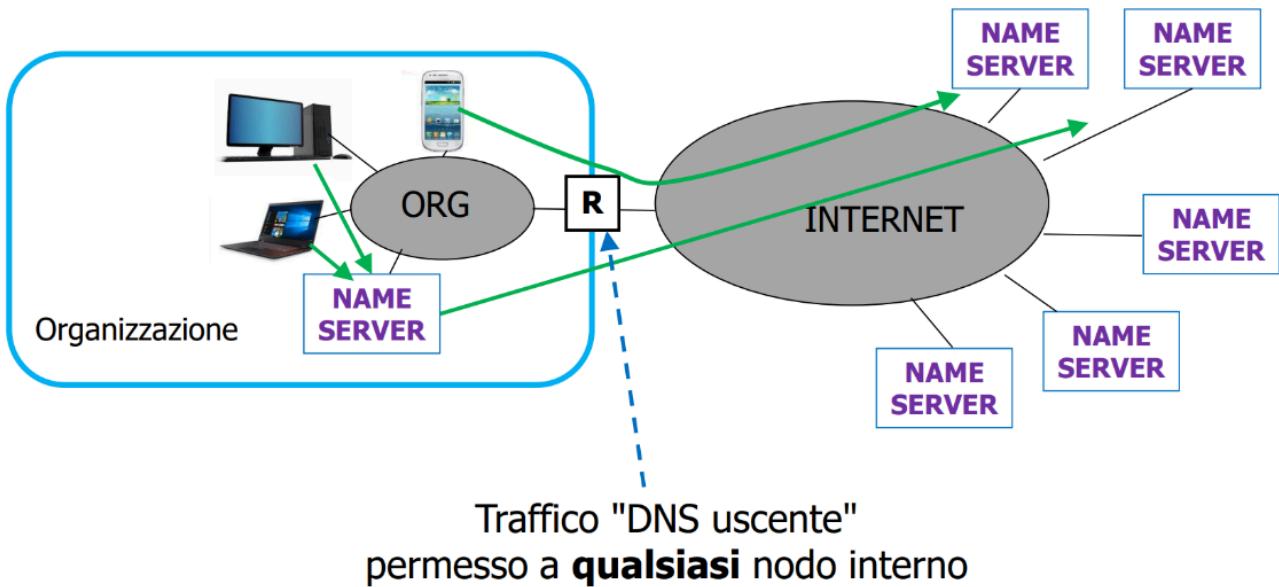


I motivi per cui si sceglie il *Caso 1* sono molteplici, tra cui:

- Motivi di prestazioni: *DNS Caching*
- Motivi di cybersecurity: Blocciamo risoluzione dei nomi "*associati ad attacchi*", ovvero che sono noti a distribuire malware o a comunicare con nodi che hanno malware
- Motivi strategici: vedere gli aspetti strategici dell'infrastruttura DNS! (Aspetti Strategici del DNS)

Caso 2 (tipico Internet Provider)

- Come *Caso 1*, solo che si permette di configurare il *proprio host* per usare un *name server esterno* (come il DNS della Google). Quindi, il router di frontiera non blocca niente
- Permettiamo traffico "*DNS uscente*" a qualsiasi host dell'organizzazione



Osservazione. Come mai alcuni offrono il proprio name server come DNS, utilizzabile da chiunque? Facendo ciò consumano un numero elevato di risorse...

- Il motivo più realistico è quello di *guadagnare più denaro* raccogliendo i dati sugli interessi degli utenti, che permette alle organizzazioni di eseguire operazioni profittabili (i.e. pubblicità mirata).

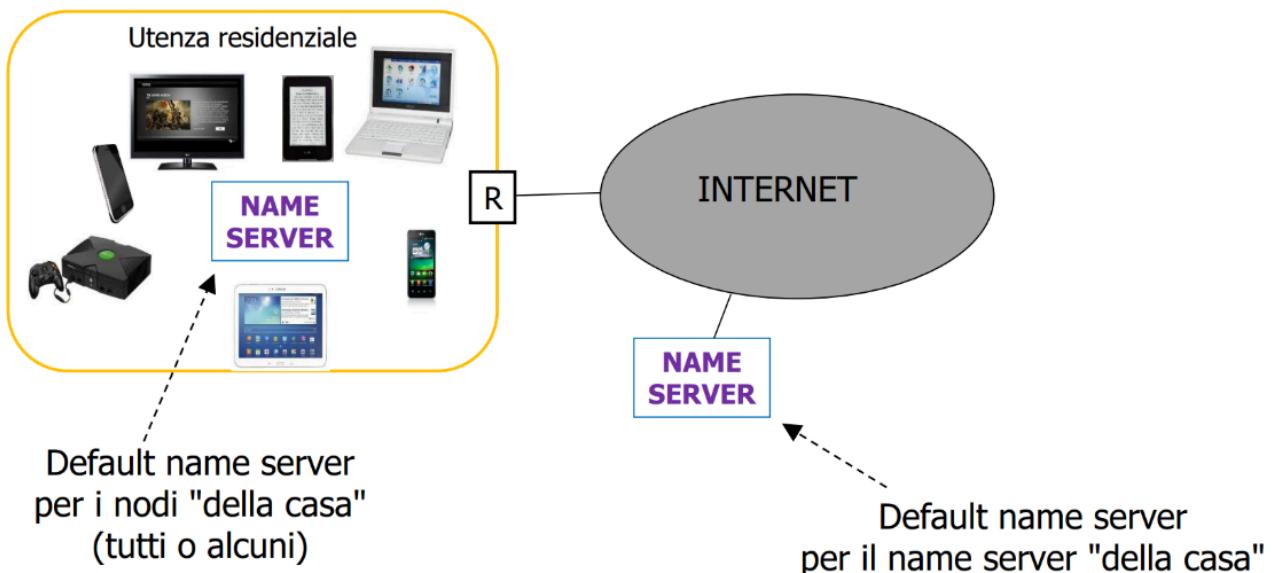
 X

2. Altri scenari DNS

Caso 3 (DNS "della casa")

In certi casi è possibile installare un *NS* sul dispositivo personale e configugarlo.

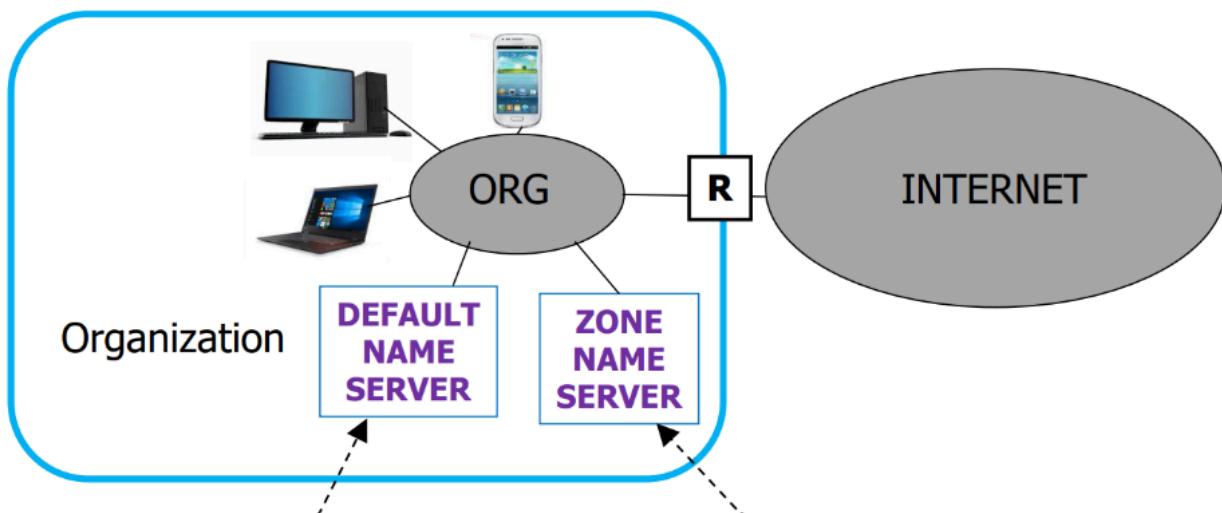
- *Esempio:* Su linux si può usare **dnsmasq**



Caso 4 (Recursive e Authoritative DNS)

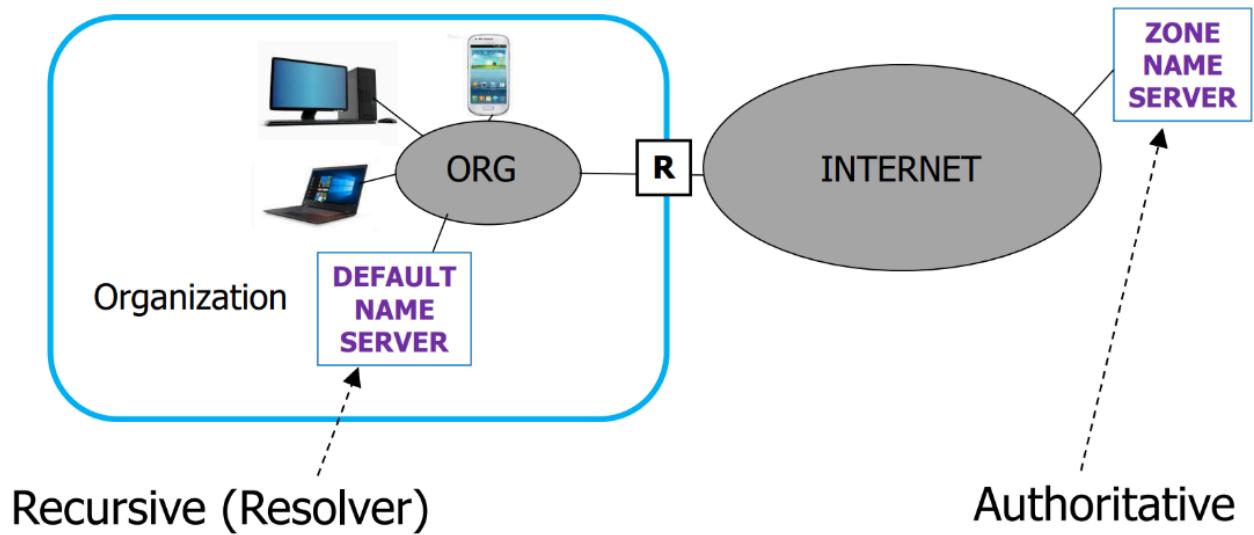
Avvolte all'interno dell'organizzazione possiamo avere *due DNS* distinti, una per *tutti i nodi interni* e l'altra contiene gli *RR* dell'organizzazione (vedremo dopo cos'è un RR, intuitivamente è una riga della mappatura name \mapsto value).

- Il primo si chiama *Recursive Name Server* e ottiene i RR richiesti dai nodi interni
- Il secondo si chiama *Authoritative Name Server* e contiene i RR della zona



Caso 4.1.

In certi casi è possibile che l'Authoritative NS si trovi all'esterno dell'organizzazione (caso comune: web hosting)



Name Resolution

X

Processo di Name Resolution. Local Resolver: precisazioni sul processo della ricerca di indirizzi IP, definizione di local resolver. Schema logico dei local resolver. Presenza RR in Local Resolver: Caching e hosts file.

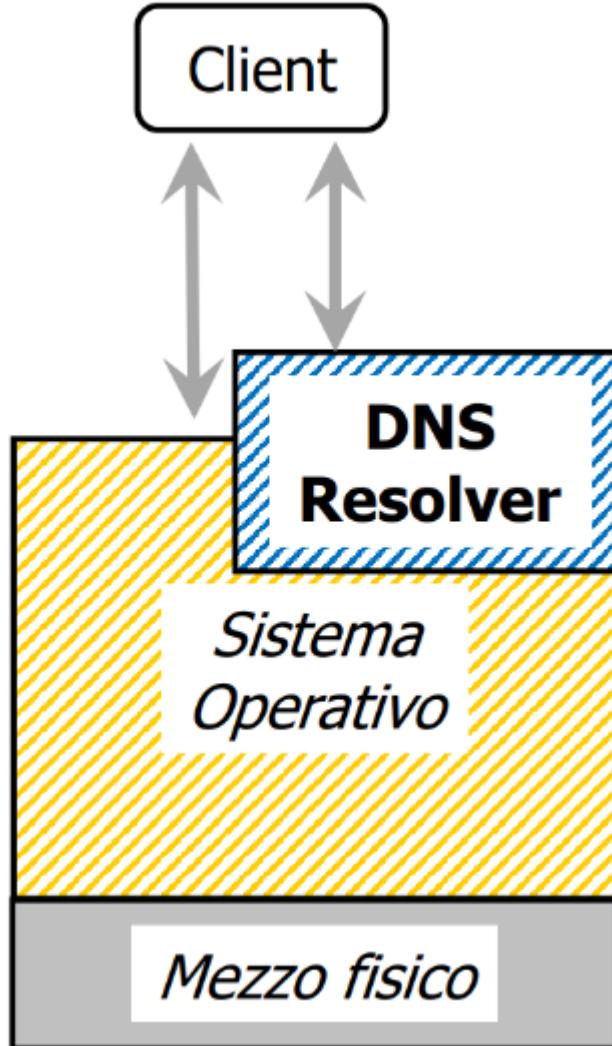
X

0. Voci correlate

- Ricerca di Indirizzi IP
- Name Server

1. Local Resolver e Name Resolution

Prima supponevamo che per trovare l'*indirizzo IP* associato ad un *calcolatore*, si andava a contattare direttamente il proprio *default name server*. Tuttavia, nella realtà abbiamo più componenti nel sistema operativo: si ha il *modulo local resolver* contenente delle funzioni che servono per effettuare il *name resolution*.

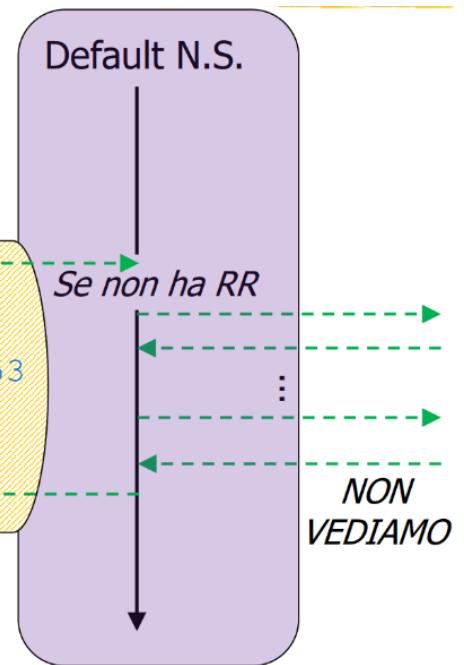
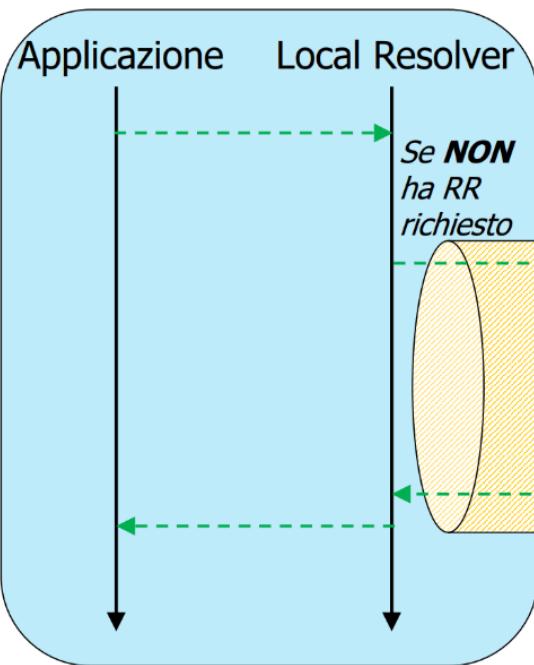
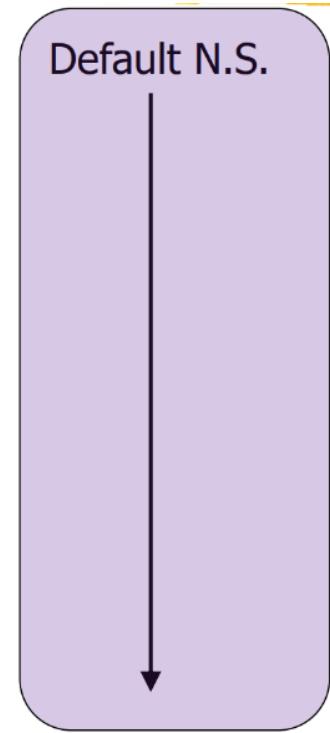
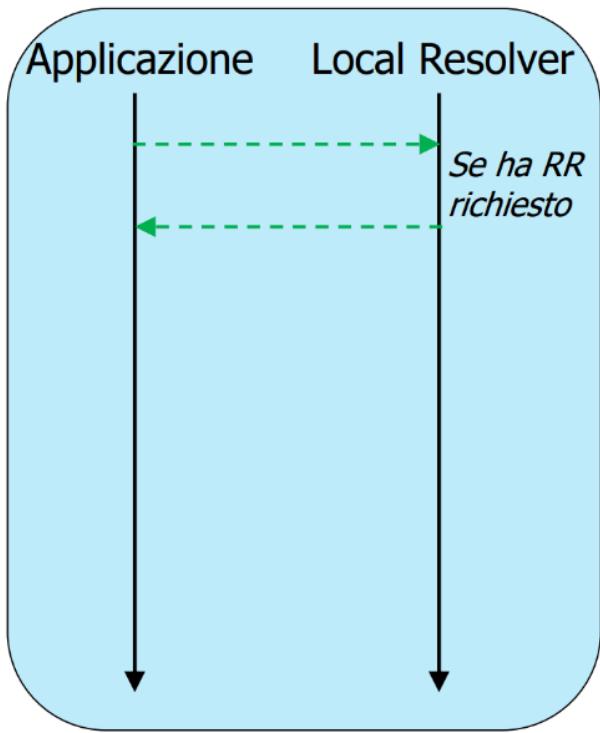


Il *name resolution* segue questo schema logico:

1. Se si conosce il *RR*, ritornare tale valore
2. Altrimenti contattare il proprio *DNS* per cercare l'*RR*

Il *local resolver* può già conoscere l'*RR* nelle seguenti casistiche:

- Ogni resolver *memorizza nella cache* per un po' di tempo gli RR ricevuti (DNS Caching)
- Si trova già la risposta nel *Hosts file*, ossia un file nel sistema operativo che traduce alcuni nomi in indirizzi IP



Cenni all'Implementazione DNS

X

Cenni all'implementazione storica del DNS. Domande: a quali DNS chiedere, come conoscere i loro indirizzi IP, come sapere che il nome non esiste, come posso creare/modificare indirizzi IP. Aspetto storico: 1985, Jon Postel. Requisiti per il sistema DNS.

X

0. Voci correlate

- Name Server

1. Requisiti dei Domain Name Server

Q. Quali proprietà deve avere un domain name server? Come faccio sapere quali DNS contattare, come conosco i loro indirizzi IP? Come raggiungo la certezza che un nome non esista? Se volessi modificare le tabelle sui DNS? Quali DNS devo modificare, e posso?

Per poter rispondere alle domande poste in una maniera soddisfacente, i DNS dovranno principalmente rispettare i seguenti requisiti:

Scala: Il sistema dev'essere in grado di "*scalare bene*", in quanto si ha a che fare con tabelle contenenti centinaia di milioni di righe (circa 4 miliardi di indirizzi IP possibili totali!)

Performance: L'accesso dev'essere chiaramente *sufficientemente vecole*, anche nel caso di ricerca fallita

Robustezza: Il sistema dev'essere accessibile anche in caso di *guasti*, quindi non ci dev'essere il c.d. "*single point of failure*".

Autonomia Amministrativa: Ogni organizzazione deve gestire "*i propri RR*" in autonomia, senza dover chiedere a nessuno.

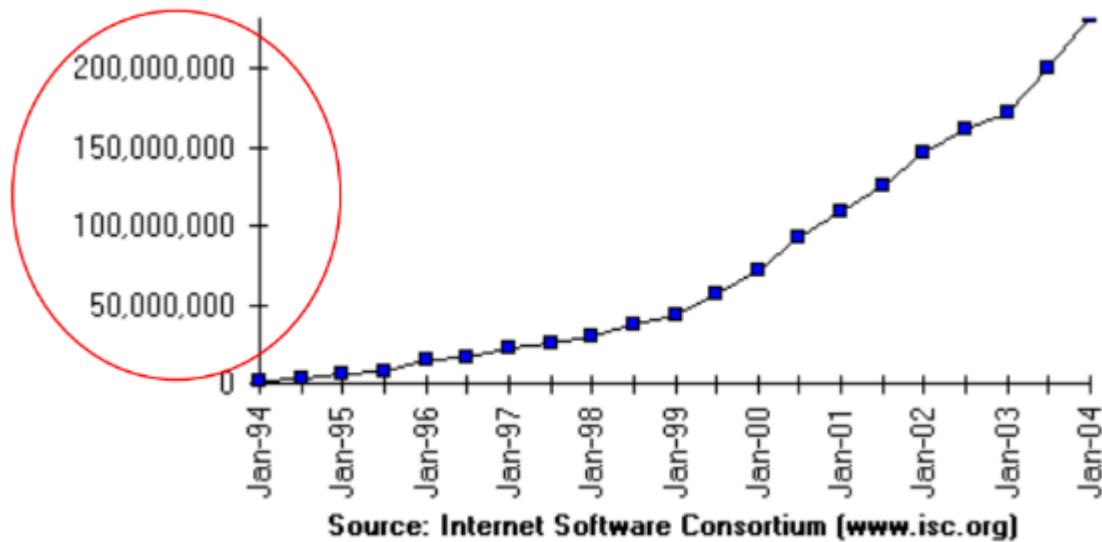
X

2. Cenni Storici al sistema DNS

1985. Jon Postel manteneva uno "*host file*" e ogni nodo aveva una *copia locale* dell'host file. Ogni modifica doveva essere comunicata a Jon Postel e poi essere ricopiata in tutti i nodi internet

Essendo che ai tempi i calcolatori erano *pochi*, questo sistema era fattibile.

1987. Primi protocolli che descrivono il DNS, RFC 1034 e 1035, che oggi funzionano ancora. La si considera come un "*miracolo*" in quanto dopo 40 anni il sistema è ancora in grado di funzionare, nonostante i "*carichi*" siano cresciuti in 6-7 ordini di grandezza



Aspetti Strategici del DNS

X

Criticità dell'infrastruttura DNS, e aspetti strategici dell'infrastruttura DNS.

X

0. Voci correlate

- Name Server

1. Criticità dell'Infrastruttua DNS

Oggi giorno, il *l'infrastruttura DNS* è una infrastruttura *enormemente importante*, una *infrastruttura critica*. Possiamo pensare che l'infrastruttura operi un ruolo analogo agli acquedotti o aeroporti nella società odierna.

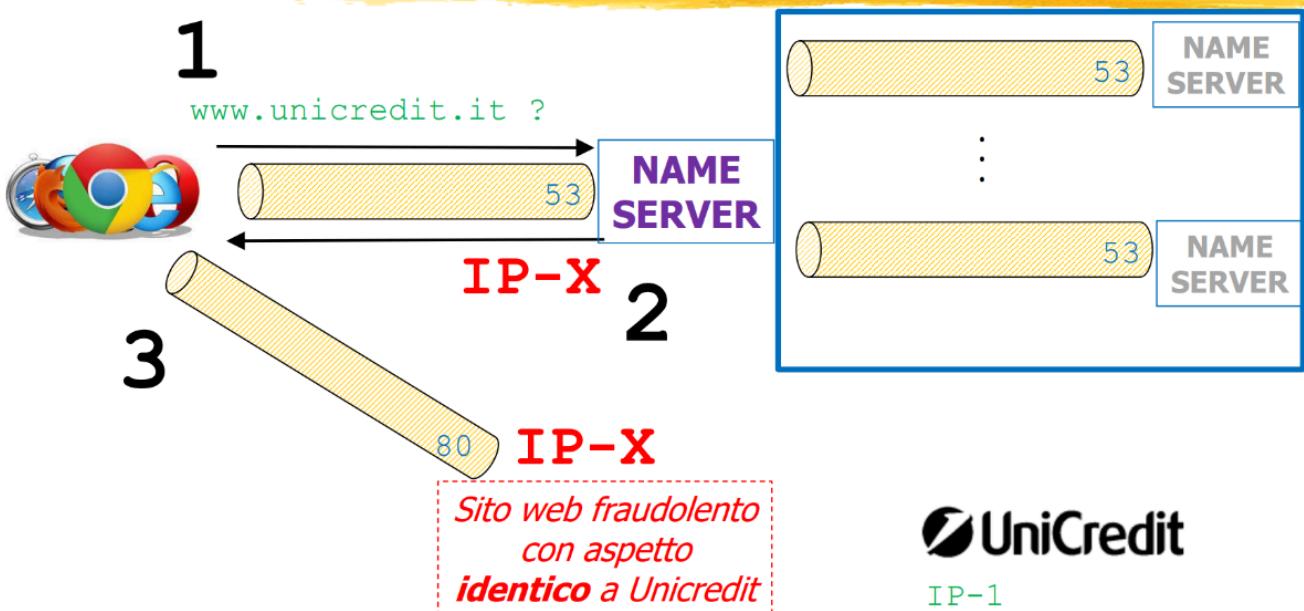
Pensiamo a due esempi che ne esemplificano la criticità:

Availability: Se non fosse più possibile tradurre i nomi che terminano con **.it**, come **inps.it**, le conseguenze sarebbero gravissime. Questo è un problema reale, con possibili *guasti* o *attacchi informatici mirati*

Hijacking: In certi casi è possibile *"manipolare"* un server DNS per ritornare *"indirizzi IP fasulli"*. Ad esempio, l'utente chiede il sito della sua banca; il DNS *"malevolo"* gli ritorna l'indirizzo IP di un sito che assomigli alla banca ma è un fake che raccoglie i dati. Anche questo è un problema reale, con tantissime cause possibili, tra cui:

- Attaccante modifica il contenuto a insaputa dell'amministratore
- L'amministratore è l'attaccante stesso
- L'attaccante modifica la configurazione DNS dell'utente, con tanti modi

Questo è un scenario realmente accaduto tante volte, coinvolgendo ad esempio il registrar GoDaddy e delle piattaforma di criptovalute.



Tuttavia, nel corso, assumeremo il seguente principio:

- Ogni nodo si comporta "*onestamente*"

Questo è dovuto al fatto che *tutti i protocolli Internet* seguono quel principio, e ai tempi non era contemplato che ci fossero degli eventuali pericoli.

 X

2. DNS: Strumento Strategico

OSS. Notiamo che accedere a servizi per *indirizzo IP* invece che per *nome* è praticamente "*infattibile*"; ciò rende il *DNS* come un'infrastruttura indispensabile per accedere a servizi. Da ciò emergono delle caratteristiche strategiche del DNS

Censura: Un governo può imporre ad ogni *Internet Provider sul territorio nazionale* di *bloccare DNS request* per certi nomi e di *bloccare traffico DNS verso indirizzi IP esteri*.

Sorveglianza: Un governo può imporre agli *Internet Provider* di tenere traccia gli *indirizzi IP* di un utente e le sue relative richieste DNS

Profilazione: Le aziende ottengono una *conoscenza "molto accurata"* su moltissimi utenti

Resource Record

X

Definizione di Resource Record. Cenni al DNS protocol: notazione intuitiva. Type principali: A, MX e CNAME. Osservazioni sul DNS response: additional RR.

X

0. Voci correlate

- Name Server

1. Definizione di Resource Records

Definizione. Un *resource record* è una tripla composta dal:

- *Nome* che identifica l'host (il nodo)
- *Type* che "caratterizza" la tipologia del valore
- *Valore* (l'indirizzo che identifica il calcolatore)

In altre parole, le *resource records* vanno a comporre la tabella di ricerca che viene realizzata dall'infrastruttura *DNS*. Fin'ora avevamo visto gli RR di tipo *A*, che identificano gli indirizzi IP.

Osservazione. In molti casi, ai nomi degli host associamo un'organizzazione, e pensiamo che l'indirizzo IP legato al nome siano gestiti dalla medesima organizzazione. Tuttavia, in realtà non c'è *nessun legame* tra l'organizzazione associata il nome e l'indirizzo IP (valore). Per il DNS, tutto questo è *irrilevante*.

X

2. Cenni al Protocollo DNS

Per il *protocollo DNS*, useremo solo la seguente notazione intuitiva:

Request: NAME TYPE ?

Response: il Resource Record completo, se c'è; altrimenti **NXDOMAIN**, che vuol dire una roba del tipo "RR inesistente" (vedremo come mai quando tratteremo i *domini*)

- Contiene una *response record*, che contiene o la *RR* richiesta o **NXDOMAIN**
- Può contenere ulteriori *additional record* (ulteriori RR), a discrezione del name server

X

3. Type Principali

MX. Gli *RR* di tipologia **MX** identificano il *dominio di una email* al *nome di host mail server di "quel dominio email"*.

CNAME. Gli RR di tipo **CNAME** identificano un name di un host in un altro nome *equivalente*.

Definizione di Dominio

X

Definizione di dominio. DNS response per domini con più RR. Osservazione: un dominio di CNAME dovrebbe essere solo composto da RR CNAME. Domain Tree: costruzione grafica, proprietà di sottodominio. Definizione di TLD e SLD.

X

0. Voci correlate

- Resource Records

1. Dominio

Definizione. Un *dominio* è un *insieme di RR con lo stesso nome*. Quindi, ogni RR appartiene esattamente ad un dominio.

Esempio. Supponiamo di avere gli seguenti RR:

```
google.com A IP1
google.com A IP2
google.com A IP3
```

Allora ho più host con lo stesso nome, ed essi appartengono allo *stesso dominio*.

Q. Se ho un dominio con > 1 componenti, come dovrebbe rispondere un *DNS* quando gli viene chieso **NAME A ?**?

Semplicemente è tutto a discrezione del *DNS*, quindi può decidere di dare *tutti*, *alcuni* o *solo uno* e i fattori che potrebbero motivare una delle scelte sono le seguenti:

- Scegliere posizioni geografiche più vicine
- Distribuire carichi a rotazione

Q. Se ho un dominio che contiene sia *CNAME* e non, cosa succede? Ho definizioni in conflitto...

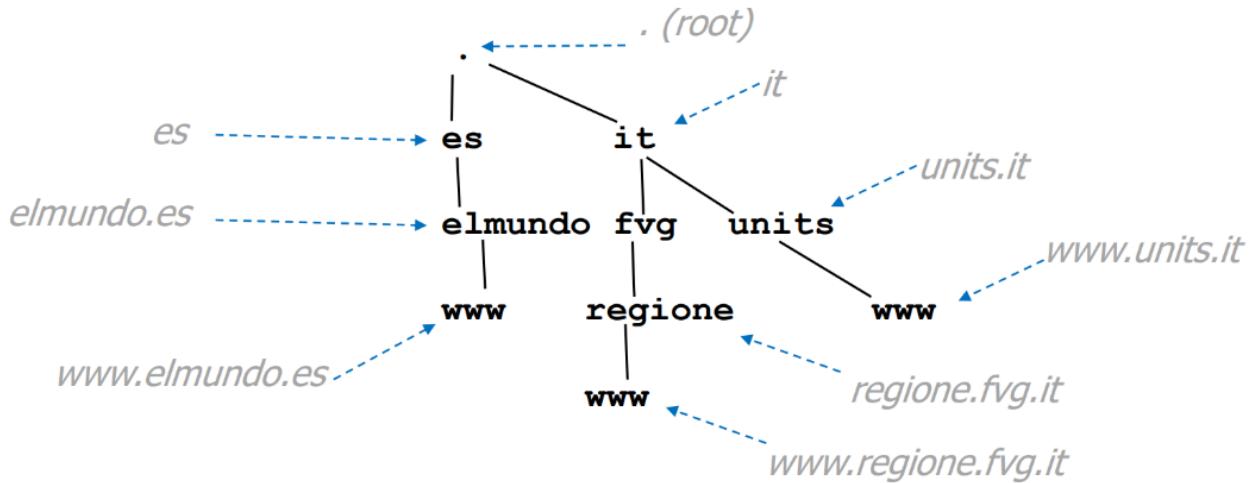
Generalmente questo non *dovrebbe accadere*, siccome per convenzione un dominio con *RR CNAME* non può contenere altri *RR di tipo diverso*. Non che sia impossibile, ma sconsigliatissima.

X

2. Domain Trees

Immagino di avere *tutti i domini esistenti*, come posso rappresentarli graficamente? Un modo per farlo è con i *domain tree*. In particolare, abbiamo un *grafo ad albero* dove:

- I *nodi* sono i *domini stessi*, e il nome è il "*label path*", ottenuto facendo la concatenazione con i loro parenti fino al nodo radice `.`.
- Gli *archi* rappresentano quindi un collegamento tra i domini (vedremo come nell'esempio)

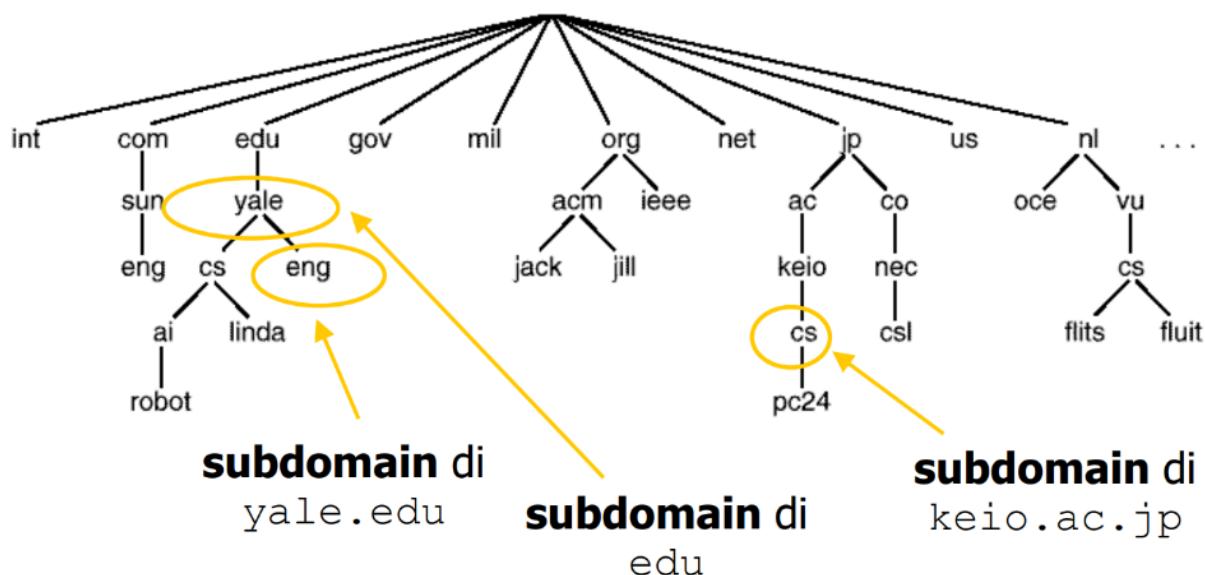


Notiamo che con questa rappresentazione *non vediamo gli RR stessi*, non sapremo dedurre *quanti e di quali tipi* RR contengono ogni nodo. Per farlo, bisognerà contattare il DNS.

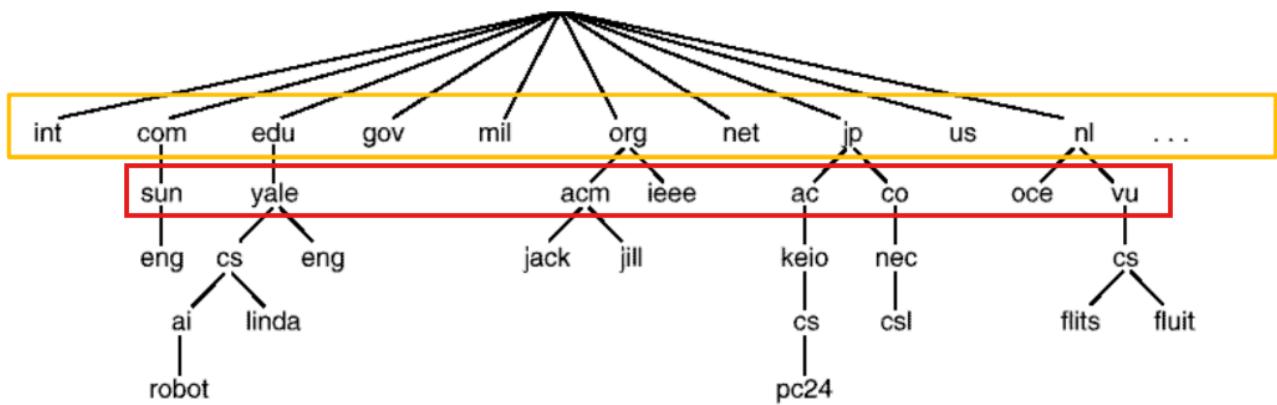
X

2. Subdomains

Principio. Ogni *dominio* è un *sottodominio* del *dominio padre*. Notiamo che ciò non implica che sono sottoinsiemi affatto, in quanto *RR di domini diversi* sono *disgiunti per definizione!*



Definizione. Un dominio si dice essere *Top-level domain* se è un subdomain della root. Se invece è subdomain di una TLD, allora si dice *Second-level domain*.



Creazione e Gestione dei Domini

X

Aspetti pratici sulla creazione e gestione di domini. Proprietari di domini: proprietà ricorsiva, delegazione di proprietà. Definizione di zona. Creazione di un dominio: caso comune e semplice. Vincoli sui nomi del dominio. Risalire al proprietario dal dominio: problemi pratici.

X

0. Voci correlate

- Definizione di Dominio

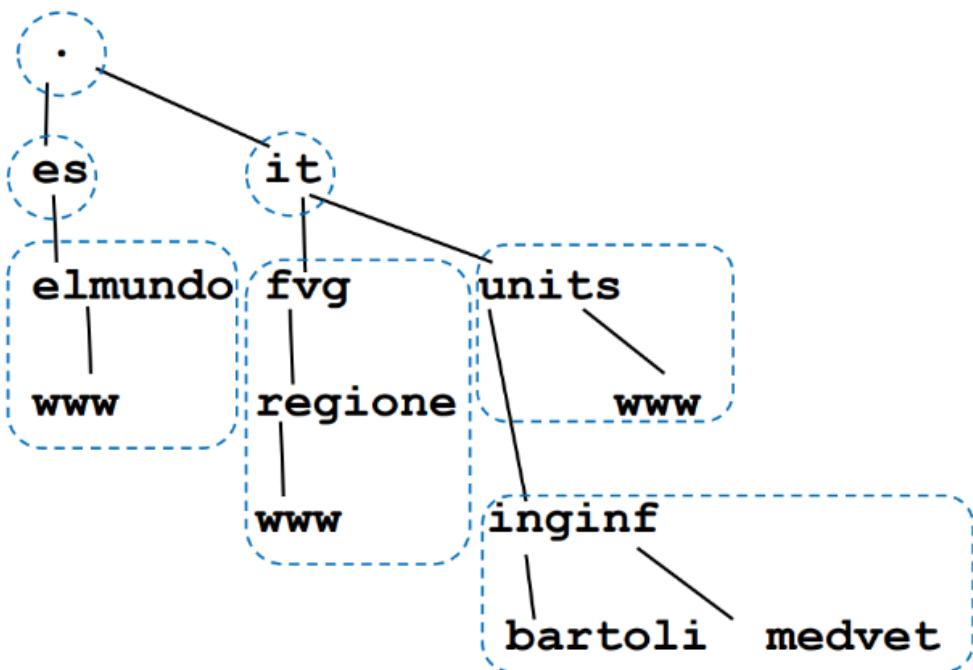
1. Proprietario di un Dominio

Ogni dominio ha un *proprietario* (individuo o entità giuridica), che ha *controllo completo su tutti gli RR*. Esiste quindi *quali* e *quanti* RR devono esistere nel dominio.

Proprietà. Ogni proprietario di un dominio è anche proprietario dei suoi *sottodomini* (ricorsiva).

Quindi, si avrebbe che il proprietario . è in realtà il proprietario di tutti i domini esistenti. Tuttavia, si ha che il *proprietario di un dominio può delegare ad altri la proprietà di un sottodominio*, spezzando quindi la ricorsività della proprietà.

Da questo si ha un *partizionamento* della *Domain Tree* in *zone*, che costituisce un "*pezzo contiguo*" con lo stesso proprietario. Fisicamente, ci sono degli *RR* che descrivono queste suddivisioni in *zone*; per quanto riguarda il proprietario, si ha un metodo per identificarlo *molto complesso* ed è oggetto della seconda parte del corso.



2. Creazione di Domini

Per creare un dominio, assumiamo il *caso più semplice* (e comune): ovvero vogliamo creare un *dominio SLD* senza la possibilità di delegare la proprietà di sottodomini.



Il procedimento per farlo è banale. Si *"acquista"* il dominio da un *"DNS provider"*, che gestisce i dettagli tecnici e commerciali con i proprietari delle *TLD*. In particolare, si occuperanno la creazione e la delegazione della proprietà dietro le quinte.

Definiamo l'acquirente del dominio il *"registrar"*, e il *"DNS provider"* il *"registrar"*.

Q. Ci sono vincoli per scegliere il *nome* del dominio?

La convenzione delle DNS dettano che le *DNS* non forniscono nessuna garanzia a riguardo, ovvero il *nome del dominio* non pongono vincoli sull'*identità del proprietario*. Quindi, ogni nome può essere acquistato da chiunque (naturalmente con delle eccezioni, alcuni nomi non sono assegnabili o sono assegnabili con procedure di autenticazione offline).

Tuttavia, è possibile *opporsi all'assegnazione di un dominio* solo dopo quando è stata fatta l'assegnazione; in questo caso si va caso per caso, in certi casi si va ad effettuare procedimenti

legali, in altri si disattiva il dominio.

Osservazione. Quindi è possibile comprare un *nome dominio* che sembri che sia associata ad un'organizzazione o persona (ma in realtà non lo sono), e quindi poter ingannare le persone sull'internet.

2.1. Risalire il Proprietario dal Dominio

In certi casi è importante poter *risalire al proprietario* dal *dominio*. Per esemplificare:

- Rilevazione attività fraudolente
- Attività illecite, come installare malware "Command and Conquer"
 - In parole brevi, un malware C&C funziona mandando segnali "*segreti*" camuffate sotto richieste DNS verso un *dominio fasullo*. In questo caso, posso codificare richieste e risposte...

Per farlo, ricordiamo che le proprietà vengono *passate*, quindi in genere il *proprietario di un dominio padre* dovrebbe *"conoscere"* l'identità del dominio *"figlio"*. In pratica, possiamo:

1. Contattare **owner(root)** e identificare **owner(TLD)**
2. Contattare **owner(TLD)** e conoscere **owner(SLD)** oppure conoscere **DNS provider(SLD)**
3. Contattare **DNS provider(SLD)** e conoscere **owner(SLD)**

Per procedere a ulteriori livelli, solo le autorità giudiziare possono iniziare procedimenti per conoscere i proprietari.

I passaggi 1., 2. e 3. possono essere fatto con i *servizi web "whois"*, che *"navigano tra delegati"* a partire da root.

Osservazione. Notiamo che in certi casi, per motivi di svariata natura, i *Domain provider* possono decidere di offuscare intenzionalmente i dati. Le autorità giudiziare possono comunque obbligare questi registrar a fornire il registrant, ma è un procedimento *lungo e complesso*... (in quanto è probabilmente un procedimento internazionalmente e non sicuramente si ha la collaborazione del registrar)

Da questa osservazione deduciamo che l'identità del *registrant* tende ad essere *facilmente offuscabile* o *falsificabile*... gli unici casi in cui l'identità è certa sono gli TLD e alcuni SLD *"importanti"*.

Concludiamo dicendo che questa infrastruttura è critica e *basata su nomi*, che rende facile l'impersonazione sull'internet.

"Web"

Nozioni sul World Wide Web

X

Introduzione al World Wide Web. Definizione del World Wide Web, Web server e Browser (web client). Definizione di URL e le sue componenti. Azioni di un browser. Intuizione del protocollo HTTP. Azioni di un web server. Trarre documenti da URL. Osservazioni sull'URL: ruolo di localizzazione, confronto con i QR code, rappresentazione degli url negli address bar e HTTP redirection.

X

0. Voci correlate

- Comunicazione tra Processi

1. Definizione di World Wide Web

Il *World Wide Web* è un *insieme di documenti "collegati" tra di loro*, in vari formati. Il più comunemente usato è *HTML* (vedremo questo aspetto nei dettagli successivamente).

Nel *Web*, chiamiamo i *server* come i *web server* ed essi forniscono il servizio di *prelevare documenti*; invece i *browser* (*web client*) prelevano e visualizzano i documenti. In altre parole, nel Web si ha che i documenti vengono inviati dai web server e vengono visualizzati nei browser.



X

2. URL

L'*URL* (*Uniform Resource Locator*) è un *identificatore univoco* (anche se, dopo vedremo che in realtà sarebbe più accurato dire che è un *localizzatore*) di ogni documento del World Wide Web e consiste in una sequenza di caratteri. In particolare, la sequenza è suddivisa in tre parti:

- **Protocollo**: il protocollo usato per comunicare col web server, ad esempio HTTP, HTTPS, FTP, eccetera...
- **Nome del Server**: a cui effettuare la richiesta. NATURALMENTE bisognerà risolvere il nome mediante il local resolver nel sistema operativo
- **Nome del Documento**: identifica il documento stesso da trovare; quindi un file

Per convenzione l'URL è formattato nel seguente modo:

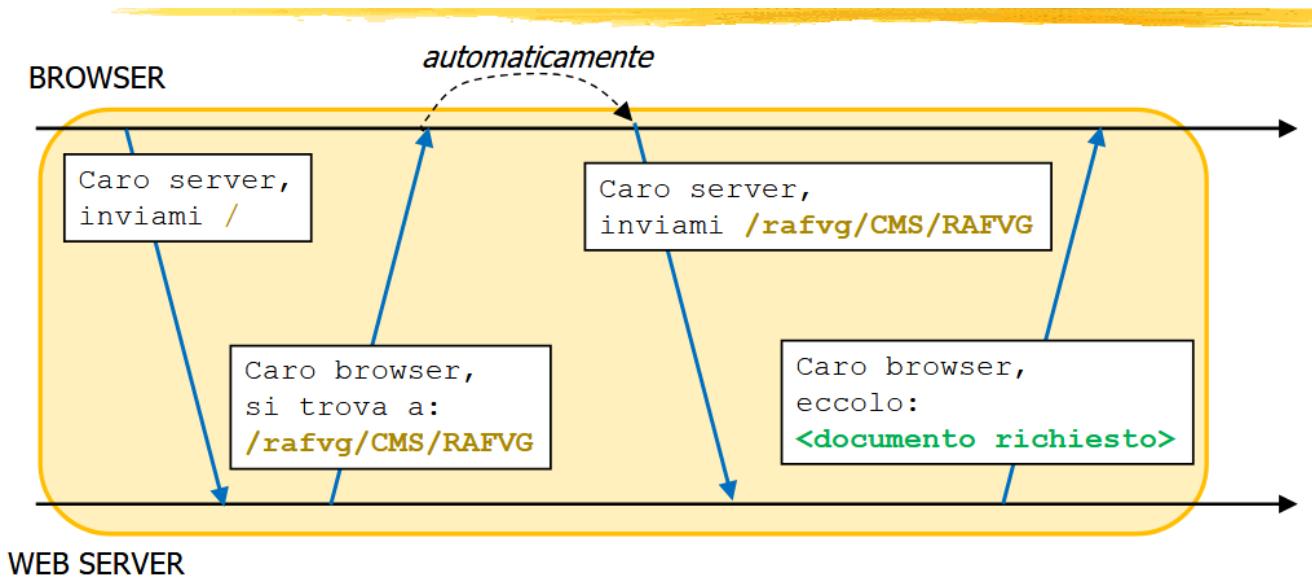
<protocol>://<server_name>/<document_name>

- Osserviamo che nel **server name** non è necessario iniziare per **www**.

QR Code. Il QR code è un'immagine che codifica una *sequenza di caratteri*, e la sua tecnica di codifica è *"robusta"* rispetto alle possibili trasformazioni di gruppo nelle immagini. Tipicamente i QR code vengono utilizzati per codificare gli URL; tuttavia ciò potrebbe essere pericoloso, in quanto l'occhio umano non è in grado di trarre un'idea dal QR code. Almeno, con gli URL si ha un'idea vaga!

Address Bar. Gli address bar dei browser sono dei campi che contengono una rappresentazione dell'URL. Nel passato gli URL venivano rappresentati *esattamente*, oggi invece si ha una *"rappresentazione semplificata"* che dipende dal browser. Ad esempio, nei browser dei dispositivi cellulari, si offusca la parte del **protocol** e viene rimpiazzata con un'icona. Oppure, un'altro caso frequente è quando il **browser** inserisce il carattere **/** alla fine del document name alla fine quando non c'è.

Inoltre, notiamo che i browser potrebbero *"decidere"* di mostrare documento con URL diverso in address bar. Questo è dovuto al fenomeno di **HTTP redirection**, che vedremo bene dopo. Per ora, la illustriamo intuitivamente con il seguente diagramma.



3. Azione Processi e Cenni al Protocollo HTTP

3.1. Web Browser

Il web browser seguirà i seguenti passaggi. Dato *in configurazione* l'URL:

- Trovare l'*IP server*, chiamando la DNS resolver
- Connetersi all'IP server con la porta specificata
- Mandare una richiesta HTTP per "*chiedere il documento*"
- Ricevere la risposta
- Chiudere la connessione
- Visualizzare il documento se possibile, o fare qualcos'altro (dipende dal tipo di documento!)

Osservazione. Notiamo che quindi *un documento* richiedere una *coppia* di *HTTP request* e *HTTP response*. Tuttavia, in realtà certe tipologie di documenti possono essere "*mandati a pezzi*", come nel caso di streaming o trasferimento di file più grandi.

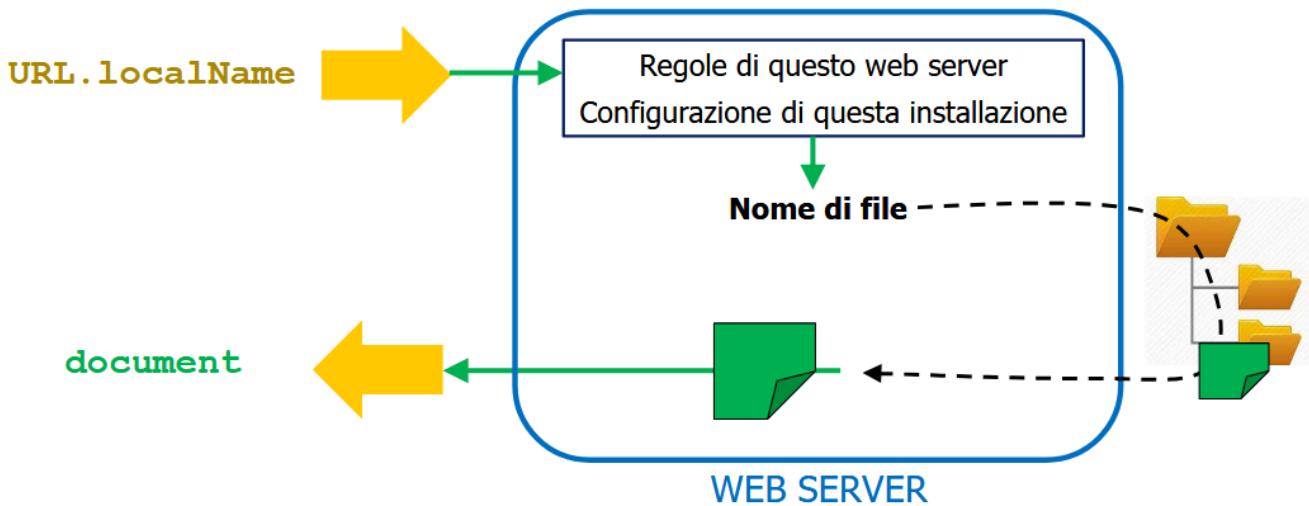
3.2. Azioni Web Server

Il web server seguirà i seguenti passaggi:

- Creare un socket e aspettare connessioni
- Quando ne riceve una:
 - Aprire il communication socket
 - Ottenere il *document name* specificato nell'URL
 - Costruire una risposta, ad esempio nel caso positivo costruire la risposta che contiene il documento
 - Inviare risposta

Osservazione. Notiamo che il passaggio di *associare il document name al documento effettivo* non è un passaggio banale, per cui esistono *molte possibilità*. Questo processo dipende da come viene configurato ogni web server, e non ci sono dei standard particolari. Comunemente si ha il *caso intuitivo*, ovvero il *document name* è interpretato come un *path relativo* su una directory del web server. Ci sono altre convenzioni aggiuntive, tra cui:

- Se non esiste l'estensione, allora viene interpretata come un file html
- Se è una directory vado a cercare il file **index.html** della directory
- Se non c'è niente (quindi c'è solo il , cerco **index.html**



Notiamo che, come conseguenza, si ha che in realtà l'*URL* è un *localizzatore*: infatti, ogni volta che sposto il file il suo URL deve cambiare.

3.3. Intuizione del Protocollo HTML

Il protocollo HTML è un protocollo text-based. Per inviare richieste o risposte, basta formattare il "*messaggio*" nei seguenti modi:

Richiesta:

```
GET <URL> HTTP/1.1
<varie righe di testo>
<linea vuota>
```

Risposta:

```
HTTP 1.1 <codice_esito>
<varie righe di testo>
<linea vuota>
<documento richiesto (se c'è)>
```

Le "*varie righe di testo*" servono per facilitare le interpretazioni nelle richieste o risposte.

Vedremo dopo cosa precisamente sono...

Struttura dei Contenuti Web

X

Struttura dei contenuti Web. Tre tipologie di file: HTML, CSS e Javascript. Metodi per hostare web server in pratica. Cenni a HTML, CSS e Javascript.

X

0. Voci correlate

- Nozioni sul World Wide Web

1. Formato dei Contenuti Web

Il contenuto dei documenti visualizzati dai browser può essere diviso in tre parti:

- *HTML*: Contenuto e struttura
- *CSS*: Stile (contenuto VISIVO)
- *JavaScript*: Azioni

HTML e CSS sono dei linguaggi "*markdown*", JavaScript è invece un vero e proprio linguaggio di programmazione (maledetto e complicatissimo...)

Per hostare un *web server*, abbiamo due metodi:

- *Self-hosting*: sul "nostro PC", accedendo il nome "localhost" oppure l'IP 127.0.0.1
- *Remoto*: Un "vero e proprio" sito web, tuttavia è necessario avere un nome DNS da qualche parte

Adesso vediamo un paio di cenni ad ogni "*parte logica*" di un documento, come si struttura.

X

2. HTML

HTML è un linguaggio per descrivere "*simple structured documents with in-lined graphics*" e ipertesti, immagini, eccetera...

Come *visualizzatore HTML* si può semplicemente usare un *browser*, oppure anche l'interfaccia di un *IDE*. Un visualizzatore HTML semplicemente interpreta il documento e nè da un'interpretazione visiva.

HTML è formato da due elementi: *testo* (content) e *tag* (structure). Ogni tag fornisce delle informazioni aggiuntive sul testo, tra cui la *struttura* o la sua *formattazione*. I tag vanno *aperti* e *chiusi*.

```
<!doctype html>
<html>
  <head>
    <title>This is the title of the webpage!</title>
  </head>
  <body>
    <p>This is an example paragraph. Anything in the
    <strong>body</strong> tag will appear on the page, just like this
    <strong>p</strong> tag and its contents.</p>
  </body>
</html>
```

Come si vede nell'esempio, ogni documento è strutturato in due parti:

- Inizia l'**head** e contiene il **titolo** e altri elementi "**background**" (come script)
- Poi c'è il **body** che contiene tutto il contenuto della pagina, quindi titoli, testo, liste, eccetera...

Inoltre, ogni **tag** può contenere uno o più **attributi**. Per sapere *quali attributi* e a quali valori assegnare, bisogna consultare l'RFC apposita.

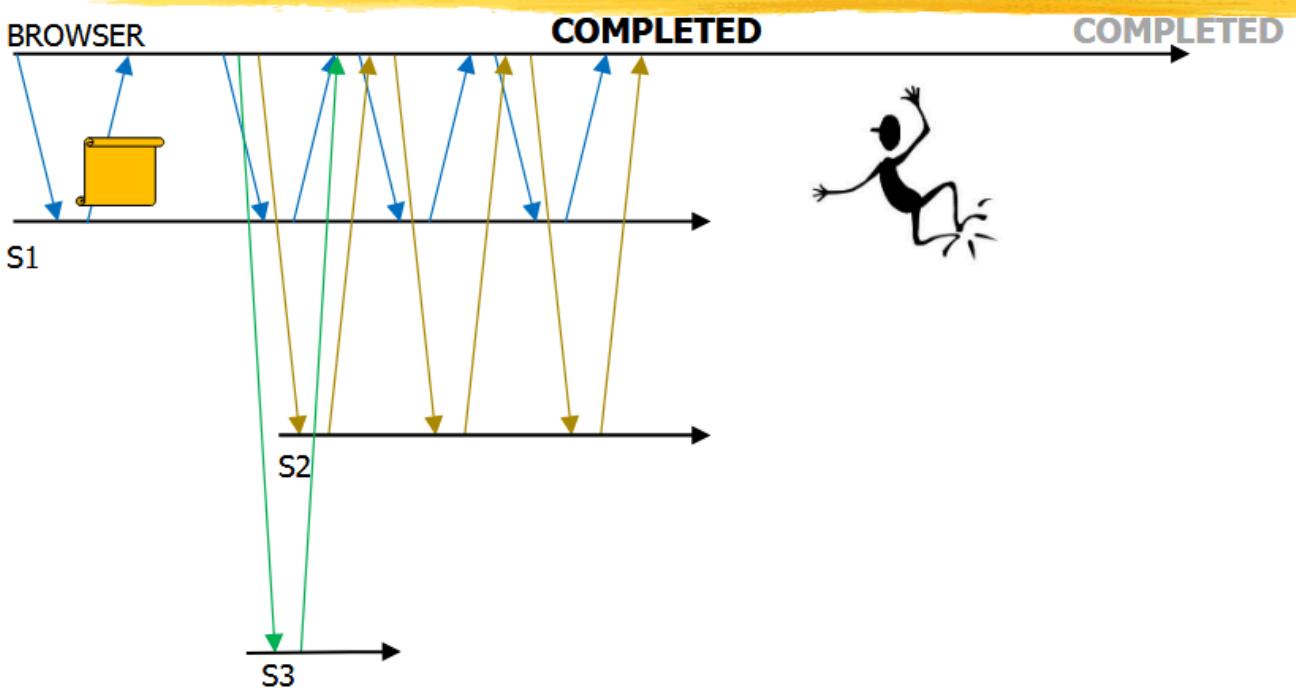
2.1. Tag Importanti

Vediamo alcuni tag importanti.

Hyperlink. Il tag ` ... ` è il **tag** che va a creare un **hyperlink**, ovvero un collegamento ad un altro sito esterno sul web.

Image. `` va a inserire un'immagine nel documento. Osserviamo che ogni volta che incontriamo un tag del genere, il web browser DEVE prelevare l'immagine effettuando ulteriori richieste a web server. Teoricamente è possibile inserirli direttamente, ma non lo faremo mai.

Iframe. `<iframe src='...'></src>` inserisce un altro documento nel documento mostrato. Come con le immagini, il browser deve prelevarlo tramite richieste http (e DNS).



Q. Cosa fa un browser?

Si vede il seguente pseudocodice che descrive le azioni di un browser:

- Per ogni "tab" aperta,
 - Interpretare il codice *HTML*
 - Prelevare *automaticamente* tutti i tag HTML che prevedono richieste ad altri server web, come le immagini, iframe, eccetera...
 - Ricordiamo che è necessario contattare il proprio *DNS*, poi aprire una connessione col web server, fare una request, ottenere la response e infine di chiudere la connessione
 - Eseguire script
 - ...
 - Disegnare in base ai documenti correnti

Osserviamo che l'azione vale *per ogni tab aperta* e quindi si parallelizza i processi (il come dipende da browser in browser). Inoltre, la visualizzazione dei documenti è *incrementale*, quindi carica i contenuti "poco appena".

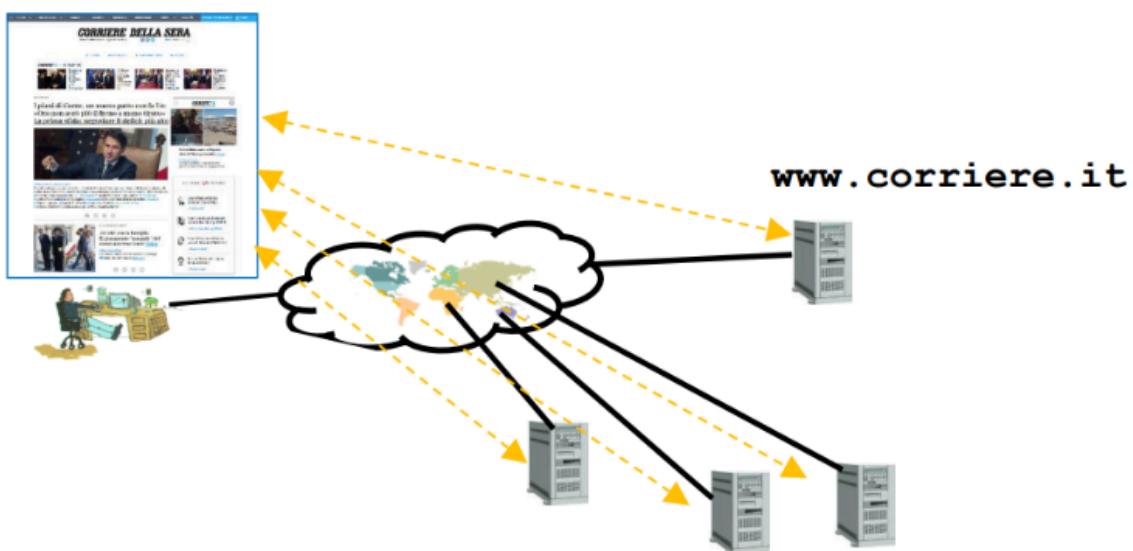
Nozione di pagina e sito web. Necessità di contattare più server per una pagina web. Chiarimenti sulle terminologie: documento/risorsa, pagina e sito. Implicazioni. Complessità dei siti web reali: come un browser gestisce un URL con solo name server. URL relativi e assoluti.

0. Voci correlate

- Nozioni sul World Wide Web

1. Pagina Web

DEFINIZIONE. Una *pagina web* è il *risultato di molti documenti prelevati automaticamente*, tra cui immagini, iframe, script/css. La barra degli indirizzi del browser contiene solo l'URL del "documento contenitore"

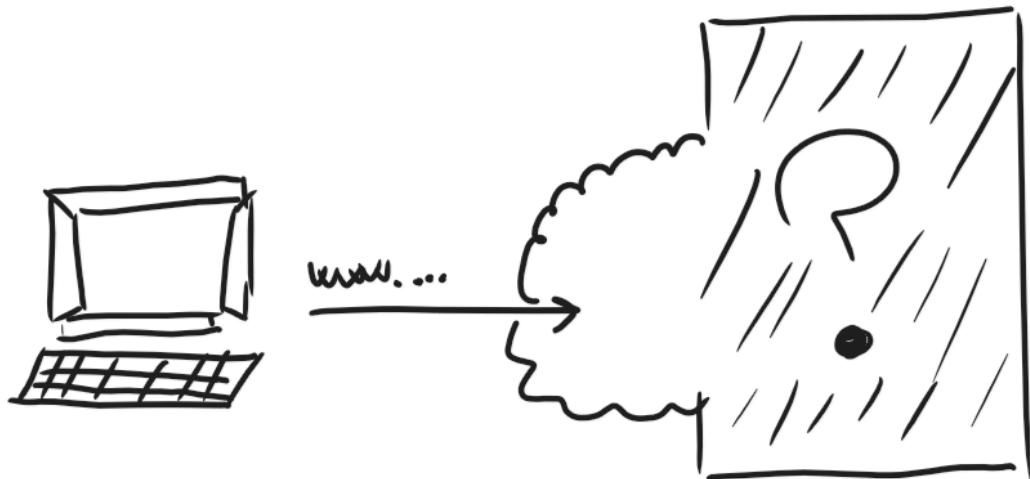


Notiamo che ciò vuol dire che una pagina web risiede su *molti web server* organizzati da enti diverse. Quindi, il browser deve sempre contattare un *insieme di server*, mai prevedibile o controllabile dal lato user.

Ciò ha delle implicazioni, e sono le seguenti:

- *Analytics*: grazie a questo sistema è possibile raccogliere informazioni su *chi va* sul sito
- *Supporto pubblicità e profiling*

Quindi, se una organizzazione è "*presente*" su molti siti web e può riconoscere lo stesso browser, allora riesce a "*conoscere tutto di noi*".



X

2. Terminologie Web

Non c'è uno *standard universalmente accettato* sulla terminologia dei contenuti web, tuttavia nel corso definiamo alcuni termini:

- *Documento/Risorsa*: Ciò che identificano gli URL
 - *Pagina*: Ciò che il browser visualizza in un tab
 - *Sito*: Insieme di pagine con lo stesso server name
-
- X
-

3. URL Relativi

Vediamo un paio di *aspetti tecnici* dei siti web, in quanto per quanto concerne gli URL.

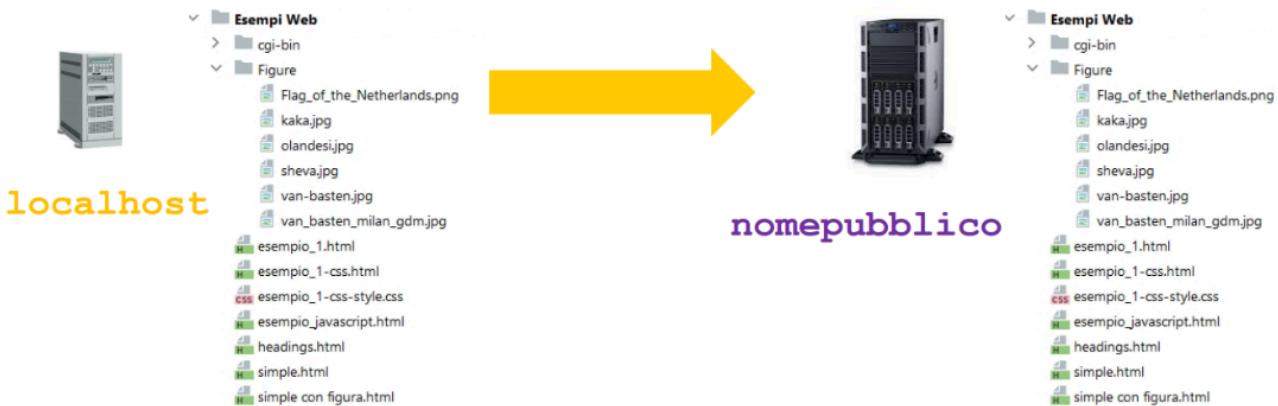
Osservazione. Quando in un file html si ha un tag con attributo `src`, è possibile mettere un URL del tipo `<tag src='myfile.x'> <\tag>`. Come fa il browser a sapere *da quale* web server prelevare? Dell'URL conosce solo la terza parte (ovvero il document name)!

Molto chiaramente si tratta di un *URL relativo*, quindi semplicemente basta copiare le prime due parti (protocollo e name server) da "*dove ho prelevato il documento*".

Vediamo un esempio dove gli *URL relativi* sono utili. Supponiamo di avere un *sito web* sul server S_1 , e poi di voler spostarlo su S_2 . Se ogni URL fosse *assoluto*, allora questo spostamento creerebbe degli problemi in quanto diventano "*invalidi*"; quindi bisognerà rilocarizzare tutto da capo...

- Si può farlo automaticamente, ma è complicato
- Farlo a mano è ancora peggio

Un classico caso di questo esempio è lo spostamento dal *server web test* al *server web produzione*: usando gli URL relativi, ci togliamo la maggior parte del lavoro da fare.



Una cosa da **ASSOLUTAMENTE NON FARE** è creare i cosiddetti "*URL relativi assoluti*", in quanto è concettualmente sbagliata.

Dettaglio. Se un URL relativo inizia con /, allora l'URL relativo è "*ancorato*" alla directory del server name.

<https://www.S2.com/docs/index.html>

```
...  
<IMG SRC="MyGif/fig.gif">  
...
```

<https://www.S2.com/docs/MyGif/fig.gif>

<https://www.S2.com/docs/index.html>

```
...  
<IMG SRC="/MyGif/fig.gif">  
...
```

<https://www.S2.com/MyGif/fig.gif>

Tutorial Sito Web

X

Gestione di siti web. Ruolo del web developer. Come fare testing: sul proprio pc o su un altro pc. Come "pubblicare" un sito web: self-hosting e acquistare un dominio DNS, affittare un calcolatore e acquistare un dominio DNS, usare un servizio di web hosting o usare un website builder. Domain custom e non custom.

X

0. Voci correlate

- Pagina e Sito Web

1. Come fare un Sito Web

Nel contesto dei siti web, ci sono due ruoli:

- Il *web developer* sviluppa il sito, quindi crea i file necessari e poi configura, esegue e mantiene il web server; anche se in realtà i due compiti sono spesso effettuati da ruoli diversi
- L'*utente* semplicemente accede al sito, quindi inserisce l'URL nel browser

Vedremo come *fare un sito web* come *web developer*

Step 1. (*Fare prove*)

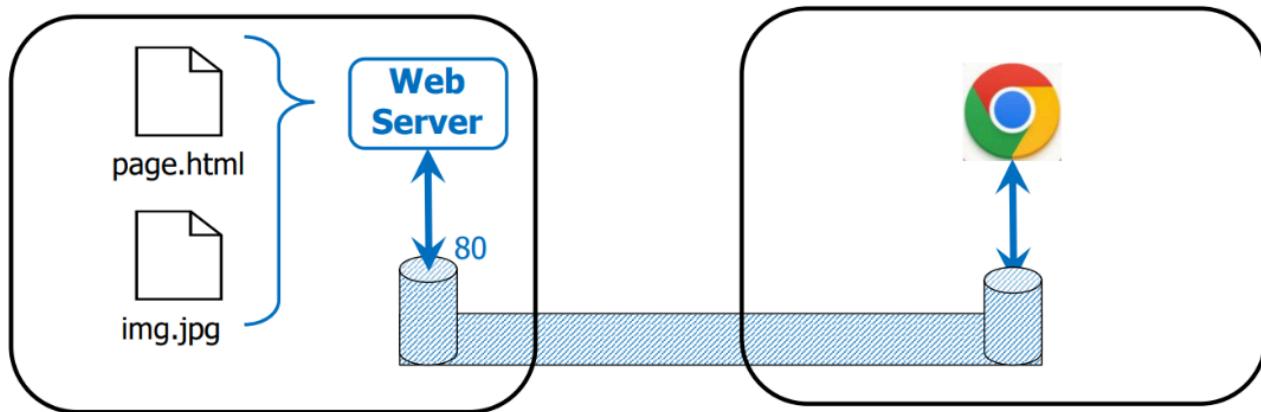
Un primo passo da fare è quello di effettuare delle prove sul sito web, quindi fingersi un utente. Ci sono più modi per fare il testing:

- *Stesso PC*, quindi il web server viene hostato localmente
- Un altro PC collegato alla *stessa rete Wi-Fi Personal*, quindi con le stesse "*credenziali*" (esempio: Eduroam no, stessa casa sì)

Vediamo il primo caso, quindi avere un *web server sul proprio PC*. Per testare, effettuare il seguente procedimento:

1. Aprire una shell
2. Posizionarsi nel folder con i file del sito
3. Lanciare il web server
4. Usare un browser e mettere url `http://localhost` oppure `http://127.0.0.1` oppure un nome a piacere sul `hosts` file

PC



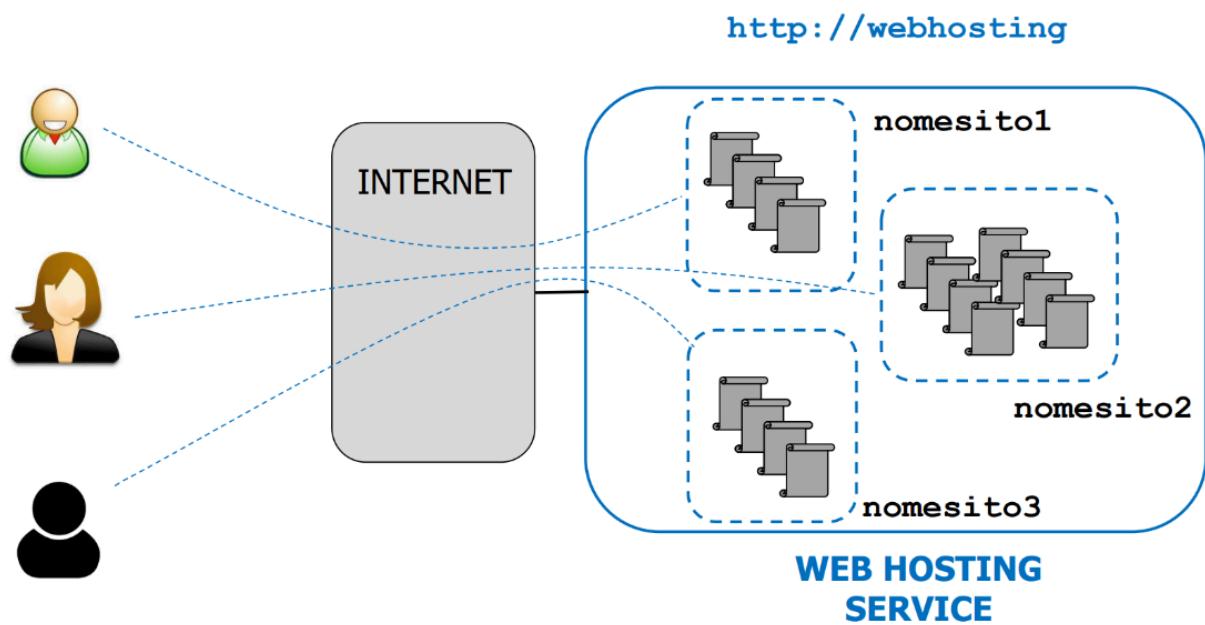
Step 2. (*Renderlo accessibile*)

Una volta soddisfatti del sito che si ha, bisogna "*pubblicarlo*" e quindi renderlo accessibile a tutti. Anche qui ci sono svariati modi per farlo:

1. Ottenere un *dominio DNS* ed eseguire il web server su un calcolatore; solitamente lo acquisto da un DNS provider
 1. Il calcolatore è *mio*
 2. Il calcolatore è *affittato* da un'azienda / ente / organizzazione / qualcuno
2. Usare un servizio *web hosting*, ossia un processo web affittato. Solitamente basta copiare i contenuti e specificare il dominio di cui sono proprietario
 1. Alcuni servizi di web hosting forniscono la possibilità di usare un *proprio dominio (custom domain)*, creando degli RR di tipo CNAME. Notiamo che comunque il "*proprietario*" del dominio è il web developer, quindi deve avere delle eventuali "collaborazioni"
 2. In altri casi il dominio viene *scelto dal servizio di web hosting (non-custom)*; in questo caso il nome è vincolato parzialmente dal servizio, siccome il dominio può essere proprietà del servizio. La parte "*comoda*" consiste nel fatto che non bisogna acquistare nessun dominio
3. Utilizzare un servizio di *website builder*, che fornisce sia il web server che un editor per modificare il sito. Quindi non bisognerà possedere nessun file di nessun tipo per creare il sito stesso.

Facciamo delle osservazioni:

- 1.1) Non posso usare questo metodo con un "*collegamento da casa*", siccome quel tipo di indirizzo IP tende a cambiare frequentemente (è *privato e dinamico*); occorrerà invece configurare un *indirizzo IP pubblico e statico*
- 2.2.) Siccome i clienti effettuando le richieste GET sullo stesso server web del servizio di web hosting, bisognerà capire a quale sito si riferisce la *terza parte dell'URL* (document name)



Client Side Scripting

X

Elementi di client-side scripting. Cosa può fare JavaScript allo browser. Dove sono gli script. Cenni a JavaScript. Osservazione: implicazioni di sicurezza (eseguire script JS sui browser), meccanismo di sandbox.

X

0. Voci correlate

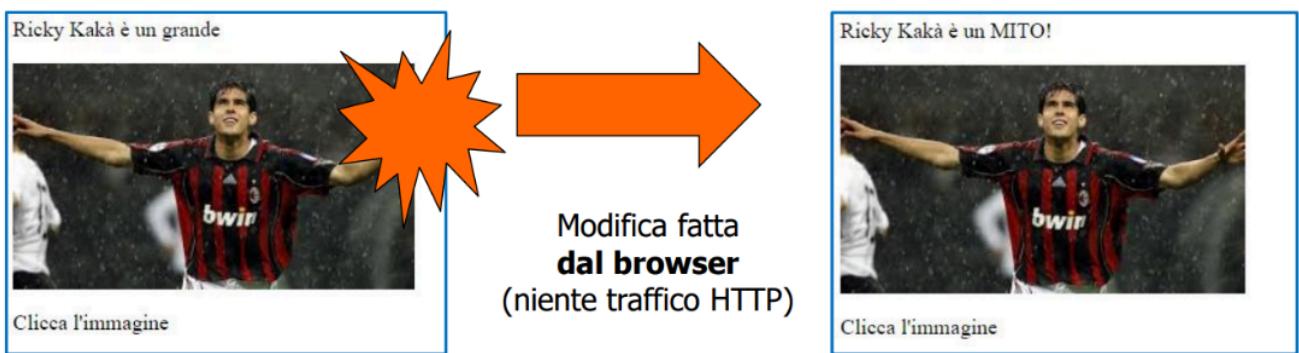
- Struttura dei Contenuti Web

1. Concetto di Client-side Script

Ricordiamoci che una *visualizzazione* ottenuta di un documento è *immutabile* e *indipendente da azioni utente*. Allora come posso aggiungerci delle interazioni, modificando la visualizzazione della pagina, senza dover generare ulteriore traffico?

Client-Side Scripting: Abbiamo degli *script* che sono in grado di *descrivere azioni* che *il browser deve effettuare*, come ad esempio aggiungere/eliminare/spostare elementi HTML, modificare attributi o proprietà CSS. Quindi, l'esecuzione degli *script* può controllare completamente l'aspetto visivo del documento!

Quindi, il concetto di *client-side scripting* consiste in creare dei *codici* da essere eseguiti (*interpretati*) dal browser, usati tipicamente per modificare l'aspetto di un documento visualizzato.



Ciò è possibile grazie al concetto di *render tree* dei browser, ricordiamoci che ogni browser esegue gli seguenti step:

- Per ogni "*tab*" aperta,
 - Interpretare il codice *HTML*
 - Prelevare *automaticamente* tutti i tag HTML che prevedono richieste ad altri server web, come le immagini, iframe, eccetera...

- Ricordiamo che è necessario contattare il proprio *DNS*, poi aprire una connessione col web server, fare una request, ottenere la response e infine di chiudere la connessione
- **Eseguire script**
- **Disegnare in base ai documenti correnti**

Negli ultimi due step si mantiene il *render tree corrente*, e gli script dicono al browser di modificare la rappresentazione nel susseguirsi di certi "eventi".

Gli script possono essere trovati in più "luoghi":

- All'interno del documento *HTML* stesso, racchiusi tra tag `<script> ... </script>`. Sono associati a "event handler" di elementi HTML specifici
- In documenti separati linkati dal documento HTML, si usa lo tag `<script src="...">`; il browser si cercherà lo script automaticamente. Quindi in realtà può trovarsi *ovunque*

X

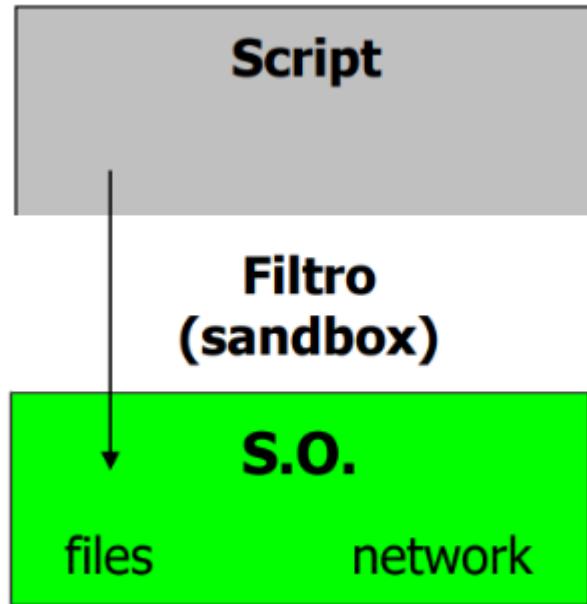
2. Java Script

Il linguaggio standard di fatto per scrivere gli script è *JavaScript*. E' un linguaggio interpretato, quindi i programmi sono sempre in *forma sorgente*; questo per rendere conveniente far eseguire gli script ai browser, siccome i loro calcolatori su cui vengono eseguiti possono essere *diversi*.

Inoltre *Java Script* può fare molte altre cose (non approfondiremo), intuitivamente si può pensare che di fatto uno script in JS è "*quasi equivalente*" ai programmi che si installano sui computer.

Ciò implica delle questioni di sicurezza, in quanto ogni volta che visualizzo una *pagina*, in realtà eseguo anche "*programmi*" sulla mia macchina e scritto da qualcun altro.

Quindi è stato designato il meccanismo di sicurezza "*Sandbox*", per cui uno script non può accedere direttamente alle risorse locali. In un certo senso, è un *filtro* per gli script JS.



Tuttavia, ciò non risolve tutto in quanto gli script JS possono comunque compromettere siti (e.g. mostrare dei dati che in realtà non corrispondono ai dati nel database); la libreria ha *controllo completo* sul web!

Documenti Statici e Dinamici

X

Documenti statici e dinamici. Remind del sistema di document retrieval. Documento dinamico: layout generale, vari punti di vista. Esempio: web app. Architettura back-end. Osservazione: nessuna differenza tra documento statico e dinamico per il browser. URL con query string.

X

0. Voci correlate

- Pagina e Sito Web
- Nozioni sul World Wide Web

1. Documenti Dinamici e Statici

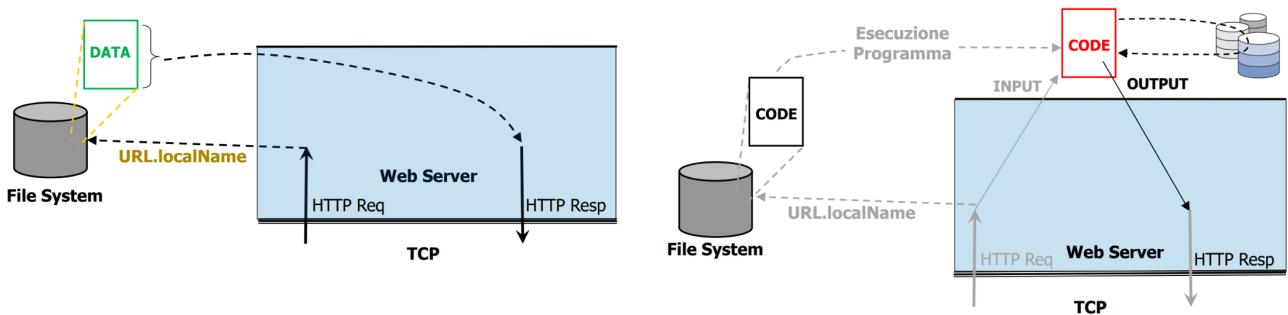
Ricordiamo che generalmente un "*document*" è un *file* sul web server, di cui il nome si ottiene da *URL.localName* con *regole* che dipendono dal web server.

Q. Quindi il sito ha preparato già *tutti i documenti possibili*? Ad esempio, è possibile che il sito di Trenitalia abbia tutti i documenti comprensivi di tutte le ricerche possibili?

No, un caso molto più comune è quello dei *documenti dinamici*; ovvero, il *document* è *costruito* da un *programma*. In particolare, *estrae parametri dalla HTTP request*, cerca delle informazioni opportune e crea il documento.



Quindi, se con un documento statico un *URL* localizza un "*documento già esistente*", invece con un *documento dinamico* la si crea alla ricezione della HTTP request.



Non ci sono convenzioni particolari per "*codici*" che creano le pagine, sono *specifici per ogni sito web*.

ESEMPIO. (*Web app*)

Nel caso delle *web app*, abbiamo che il documento è costruito da un programma indipendente dal web server e che deve seguire le convenzioni del web server. Ci sono molte tecnologie back-end per farlo, e non li vedremo. Nel corso assumiamo che esista un *programma web server* che "*fa tutto*".

Osservazione. Nel caso in cui un *web server* riceve una HTTP request per un URL che identifica sia una pagina statica che dinamica, allora essa usa le *sue regole* per decidere se tornare il contenuto statico o dinamico.

Osservazione. Notiamo che un *documento statico* non è *immutabile*, siccome posso comunque *avere degli script* che modificano il suo aspetto

Osservazione. Osserviamo che per il *browser* non importa se il documento tornato nella HTTP response è statico o dinamico; alla fine il web browser effettua le stesse zioni. Ci sono certamente degli "*indizi*" per dedurre se un documento è statico o dinamico, ma sono irrilevanti a fine pratici. Un primo esempio è quello di vedere l'estensione del local name.

Uno dei tanti indizi per capire se un documento sia statico o dinamico è quello di identificare le *URL query string*, alla fine. Essi vengono usati *tipicamente* su *web server*, e contengono dei *parametri input del programma*. Intuitivamente, la sintassi è qualcosa del tipo:

```
<protocol>://<name_server>/<my_file>?s1=t1;s2=t2;...;sn=tn
```

Di solito servono per effettuare query su database.

Protocollo HTTP

X

Protocollo HTTP. Introduzione al protocollo HTTP, le sue proprietà. Sintassi delle richieste e risposte HTTP. Concetto di headers. Osservazione: come viaggiano le trasmissioni HTTP su connessioni TCP. Approfondimenti sugli header da sapere: header host, referer, location, user agent, set-cookie/cookie, content-type/length.

X

0. Voci correlate

- Nozioni sul World Wide Web

1. Introduzione a HTTP

Il protocollo **HTTP** è estremamente complesso, ne vedremo solo un suo sottoinsieme e semplificheremo alcuni aspetti.

HTTP (*Hyper Text Transfer Protocol*, RFC 2616) è un protocollo *a richiesta/risposta* text-based in ASCII. Il protocollo è indipendente dal livello di trasporto, solitamente nel WWW si usa **TCP**; in altri casi si può usare anche **TCS/TCP**.

HTTP è uno dei protocolli più usati, infatti nel 1997 il 75% del traffico internet erano richieste HTTP; oggi ne sono circa il 98%.

X

2. HTTP Request e Response

Come accennato, le richieste HTTP sono strutturate nel seguente modo:

```
GET <URL> HTTP/1.1
<varie righe di testo>
<linea vuota>
```

Nelle "varie righe di testo" ci sono le *request headers*, ossia un insieme di linee contenenti delle coppie *Header: Value*. Ogni specifica del protocollo HTTP definisce gli insiemi di Header e Value permessi, e quali sono i loro significati.

Esempio.

- *User Agent*: Indica su quale piattaforma, dispositivo e altre informazioni sta il web browser
- *Accept Language*: Indica la preferenza della lingua

Osserviamo che quindi il *web server* costruisce il documento *anche* in base agli header in richiesta!

Adesso vediamo le risposte HTTP:

```
HTTP 1.1 <status_code> <reason_phrase>
<headers>
<linea vuota>
<documento richiesto (se c'è)>
```

La *Status Code*, accompagnata dalla *reason phrase*, indica l'esito della richiesta e sono suddivisi in 4 tipologie:

- **200**: Esito positivo, OK
- **3XX**: Redirection ad un altro sito, ovvero ha trovato il documento da un'altra parte
- **4XX**: Client error, ad esempio il client non è autorizzato a vedere certi contenuti
- **5XX**: Server error, problematiche col web server

Come si hanno gli *request header*, abbiamo gli *response header* e funzionano allo stesso modo; servono per fornire informazioni aggiuntive. Essi sono:

- Content-type
- Content-length
- Content-encoding

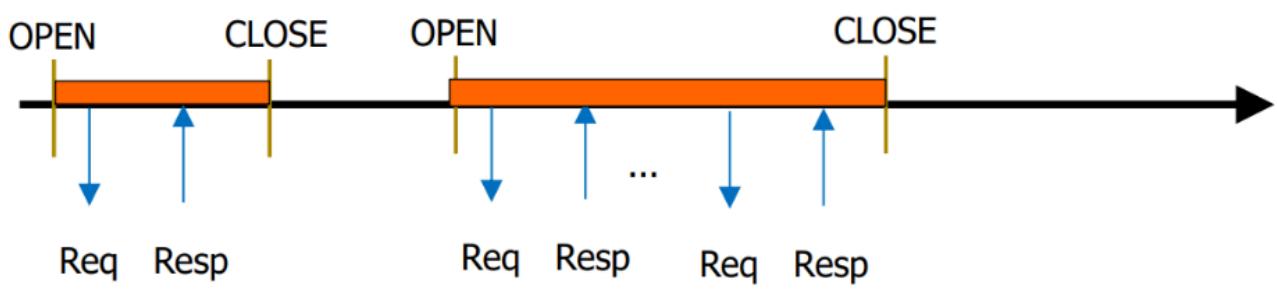
Osservazione. Nella response *non* si ha il nome del documento!

X

3. HTTP e TCP

Facciamo un breve detour su *come* viaggiano le trasmissioni HTTP su TCP:

- Ogni connessione TCP invia *una o più* coppie di HTTP request-response
 - In realtà la situazione è più complicata, siccome possono verificare altre situazioni (request pipelining, HTTP2, QUIC, ...); non li vedremo
- Si chiude quando lo decide il *Browser* o il *Server*, a loro discrezione. Se lo decide il server, il browser viene *"avvisato"* nell'ultima response con l'header *Connection*



4. Header HTTP

Adesso approfondiamo degli header da sapere

4.1. Header Host

Request Header

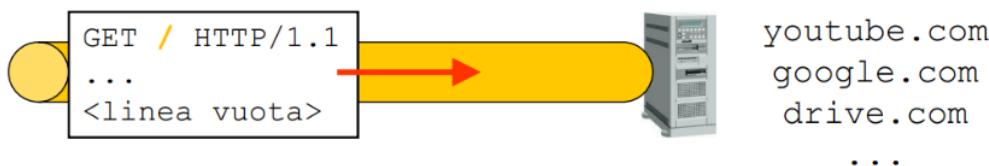
Header: Host

Value: Web Server Name (quindi la seconda parte dell'URL!)

Questo header serve per evitare alcuni casi ambigui, come i casi in cui *un web server* ha più *name server* (come ad esempio Google, Youtube e Drive); quindi per risolvere questa sorta di disambiguità, si specifica il *host name* nell'header.

Altri esempi di casistiche simili sono *web hosting* e *proxy* (vedremo i proxy dopo).

youtube.com.	299	IN	A	172.217.9.78
google.com.	299	IN	A	172.217.9.78
drive.google.com.	299	IN	A	172.217.9.78
calendar.google.com.	86399	IN	CNAME	www3.l.google.com.
www3.l.google.com.	299	IN	A	172.217.9.78



4.2. Referer

Request Header

Header: Referer

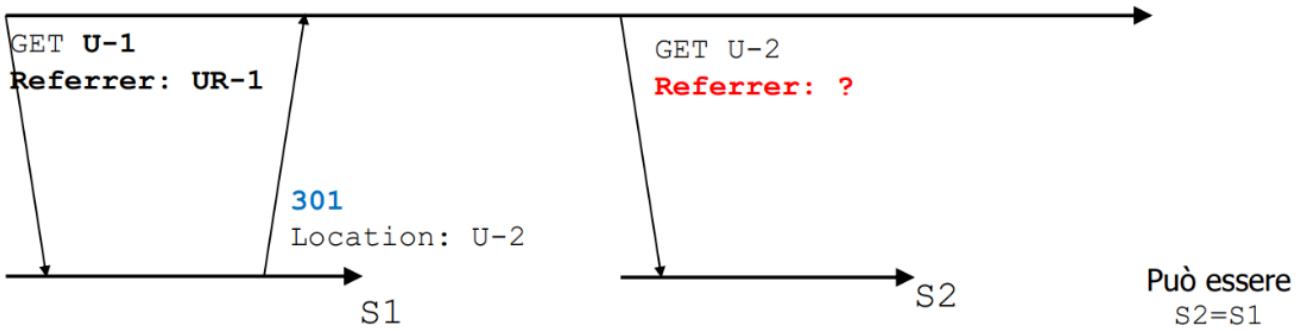
Value: URL

Il *referer* identifica il documento in cui è stato trovato l'URL richiesto, diciamo quindi l'URL che "*mi ha spinto a cercare*" il link.

(*approfondimento personale, non fidarsi!*) Questo è uno degli header più controversi; viene visto come un compromesso tra privacy e utilità fatto male.

Osservazione. Nel caso delle HTTP redirection, il referer può cambiare a seconda del browser (quindi NON specificato nel protocollo HTML):

- Dare l'URL originale
- Dare l'URL che ha "*detto*" di effettuare la redirection
- Non dare nulla, omettere quindi il referer



4.3. Location

Response Header

Header: Location

Value: URL

Questo header identifica il *"vero posto"* del documento. Questo viene utilizzato nei casi di **HTTP redirection**, infatti se il browser riceve una response con *location header* allora essa invierà automaticamente un'altra request verso l'URL specificato.

4.4. User-Agent

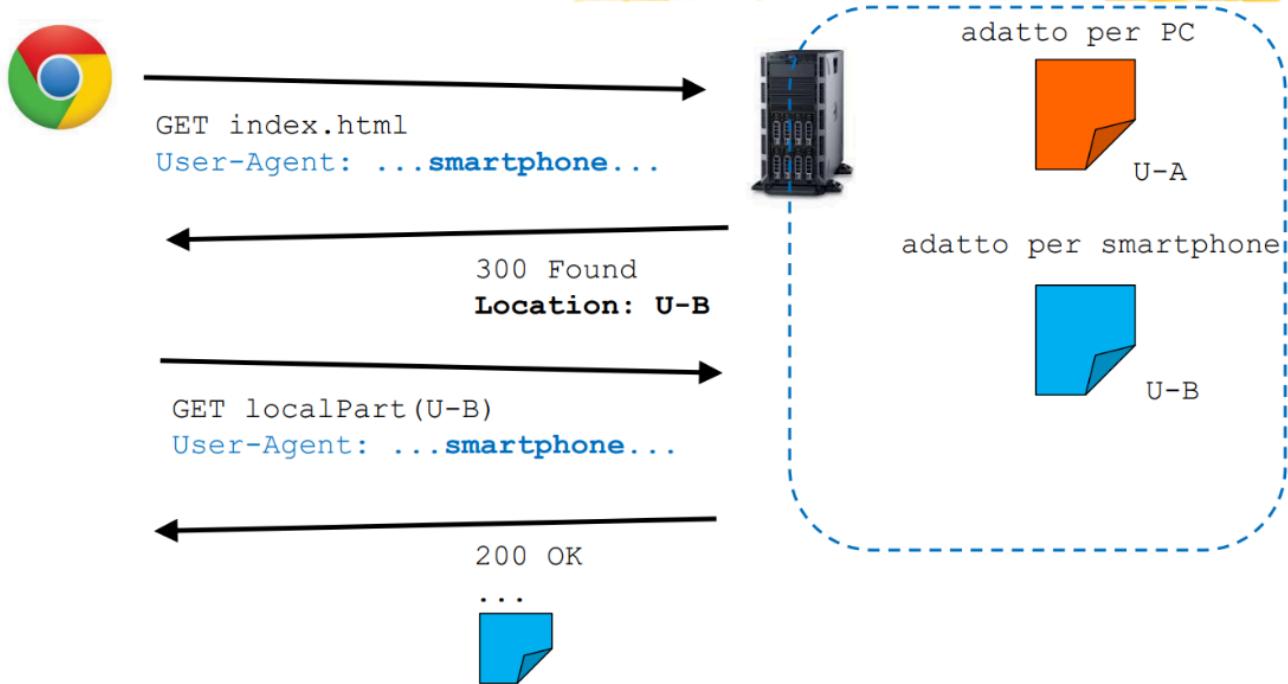
Request Header

Header: User-Agent

Value: String (formattato con la convenzione specificata)

Questo header descrive il *browser usato* e *quale dispositivo* si sta usando per effettuare la richiesta. Il web server può:

- Ignorarlo
- Rispondere con un *"documento ottimizzato"* per quel particolare browser. Ci sono due modi per farlo:
 - Mediante una *redirection*, quindi rispondere con 3XX; approccio abbastanza diffuso fino a qualche anno fa
 - Designare una *responsive web site* (quindi dinamico!), quindi di scrivere dei script CSS/Javascript opportuni per *"adattare"* la pagina allo schermo in cui si trova. Ciò non crea traffico aggiuntivo, ed è l'approccio attuale



4.5. Altri Response Header

Response Header

Header: Content-type

Value: Stringa (un formato o un insieme di formati, tipo .jpeg, .html, eccetera...)

Osservazione: Vedremo che può essere anche una Request header, vedremo in dettagli dopo...

Response Header

Header: Content-length

Value: Intero (scritto sotto forma caratteri ASCII)

Proxy HTTP

X

Proxy HTTP: definizione, dinamica reale. Motivazioni per i proxy, configurazione delle proxy.

X

0. Voci correlate

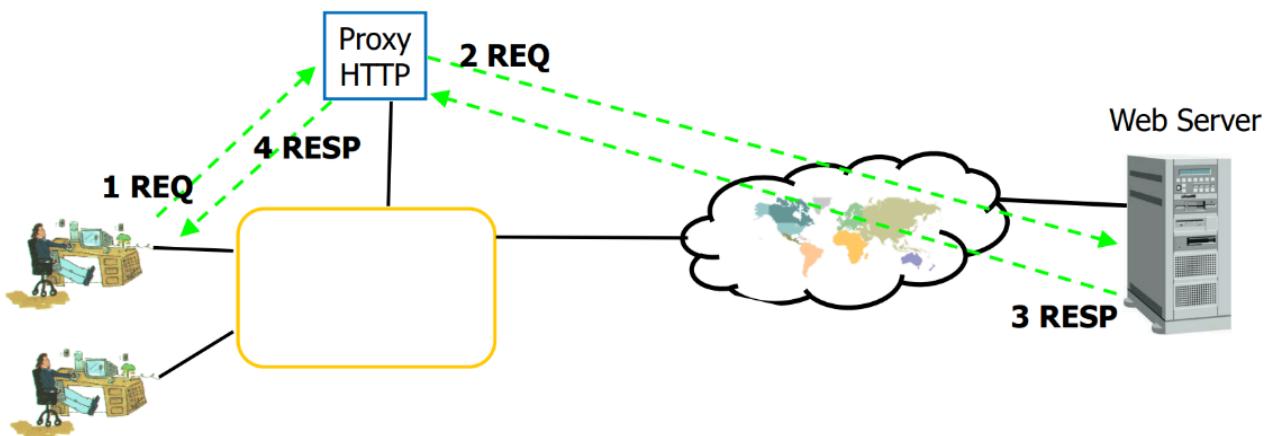
- Name Server
- Nozioni sul World Wide Web

1. Proxy HTTP

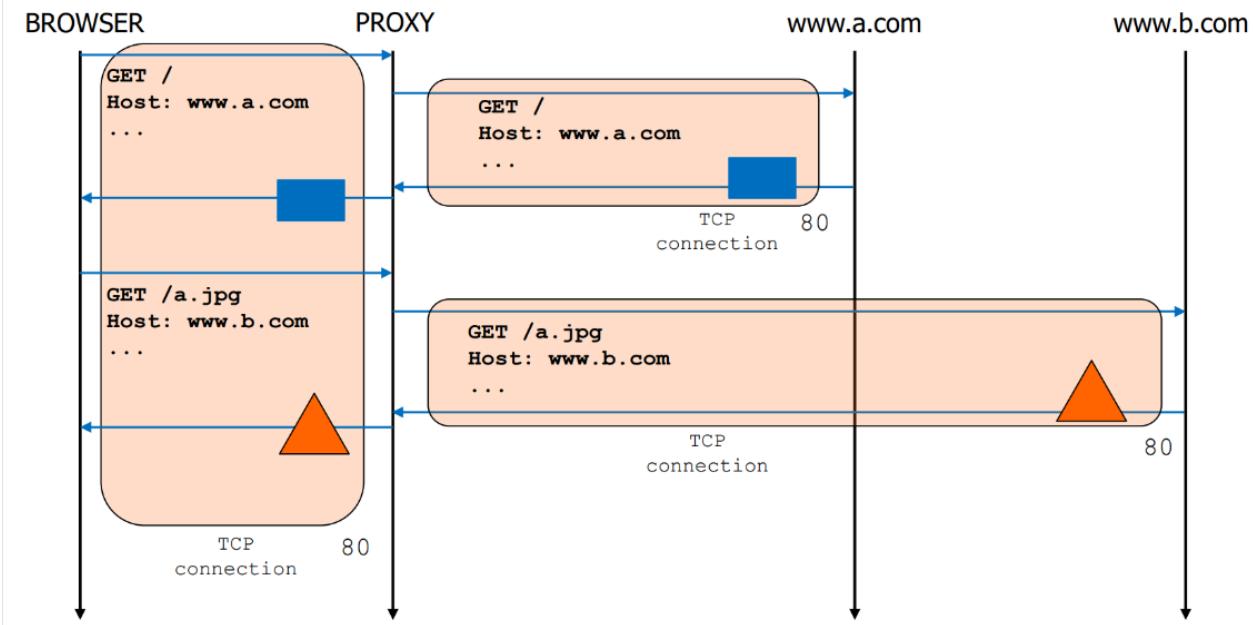
Ricordiamo che il *web server* effettua le seguenti azioni, dato un URL:

- Trova l'indirizzo IP associato alla seconda parte dell'URL (il server name)
- Si connette e comunica col web server
- ...

In realtà, molte organizzazioni non permettono il *collegamento diretto* tra *web server* e *client*. C'è un *processo intermediario* che funge da *server* per il client, e da *client* per il web server. Questo processo si chiama *proxy*.



Quindi il vero procedimento dovrebbe iniziare con la *risoluzione* del *proxy name*. Notiamo che quindi il browser non andrà mai a cercare il valore associato al name server!



Q. Perché?

Ci sono vari motivi di carattere tecnico. Ad esempio, uno dei ruoli del proxy è quello di monitorare gli URL visitati e i suoi contenuti, ed eventualmente proibirne alcuni per motivi di sicurezza. Notiamo che è indispensabile che si usi i *proxy HTTP!*

Q. Devo fare qualcosa se il web browser deve usare il proxy? Come ci assicuriamo che il browser vada a usare il proxy e non "*faccia il furbo*"?

Prima di tutto, sul *router di frontiera* blocchiamo tutti i traffici di porta 80/443 da client *non proxy*, quindi costringendo ai web browser di usare i proxy. Per quanto riguarda il web browser, o si configura direttamente il browser, o si configura il sistema operativo. Nella maggior parte dei casi, oggi la si configura in automatico.

Q. Come fa il proxy capire *a quale web server* collegarsi?

Lo si capisce dall'header host.

Invio Dati a Web Server

X

Inviare dati a web server. Procedura generale con form HTML: sintassi della tag **<form>** e usare query string. Metodo POST per inviare dati privati, sintassi. Differenza tra request GET e POST.

X

0. Voci correlate

- Pagina e Sito Web
- Client-side Scripting
- Documenti Statici e Dinamici
- Protocollo HTTP

1. Inviare Dati a Web Server

Fin'ora abbiamo *prelevato risorse* da web server. Come si effettua il viceversa, ossia *inviare informazioni* al web server?

Esempi di casi in cui è necessario l'uso:

- Verbalizzazione esami
- Email
- Acquisti
- Eccetera...

Il procedimento *generale* è come segue:

1. Inserire i *dati* su un "*campo*", specificato mediante un protocollo di invio dati (esempi: FORM, BASIC, vedremo dopo...)
2. Costruire una stringa contenente i dati
3. Inviare una HTTP request con la stringa

Per la 1., vediamo il tag FORM:

1.1. Tag Form

Tag: `<form ...> ... </form>`

Attributes:

- *Action*: URL (inserire l'URL a cui si intende di inviare i dati)
- *Method*: ??? (Vedremo dopo cosa mettere)

All'interno dello spazio tag `form`, invio più campi in cui definisco i metodi con cui si inviano i dati. Elenchiamo alcuni esempi:

- `<input type="..." id="..." name="..." value="...">`
- `<textarea id="..." name="...> ... </textarea>`

X

2. Metodi di Invio Dati

2.1. Query String

Un primo metodo semplice per *inviare dati* è quello di usare le query string. Quindi inviare una request `GET URL?S` dove `S` codifica gli input specificati.

2.2. Metodo POST

Il metodo delle query string non potrebbe essere un'ottima idea, siccome potrebbero contenere dei *dati sensibili* e le *browsing histories* sono memorizzati in più "posti"

Un altro modo è quello di inviare una richiesta di tipo `POST`, ovvero la richiesta sarà formata come il seguente:

```
POST url_name HTTP/1.1
  content_type: application/form
  content_length: ...
EMPTY LINE
Data
```

Notiamo che è sempre una *HTTP request*, ma di "*tipo diverso*". Vediamo delle differenze tra tipo GET e POST:

GET:

- Non trasporta documenti
- Caso comune
- Si aspetta delle risposte, tipicamente un documento dinamico

POST:

- Trasporta dei dati, quindi si DEVE specificare gli header `content_type` e `content_length`
- Invio informazioni raccolte nel HTML FORM

Sessioni HTTP

X

*Sessioni HTTP. Motivazione introduttiva. Definizione di sessione, implementazione con i Cookie.
Stato delle sessioni: tabella delle sessioni, variabili di sessione.*

X

0. Voci correlate

- Documenti Statici e Dinamici

1. Motivazione Introduttiva

Un documento è costruito da un *programma* che tiene conto della *HTTP request* e le precedenti inviate dallo stesso *browser e dispositivo*. Fino ad'ora, sappiamo come gestire la prima parte; e la seconda parte? Ovvero, come facciamo a realizzare un sistema che sia in grado di "*ricordare*" le request inviate precedentemente?

Vediamo dei primi approcci (che non avranno successo):

- Agganciare tutte le HTTP request provenienti dalla *stessa connessione TCP*: non funziona, siccome le connessioni TCP possono essere aperte e chiuse arbitrariamente
- Come prima ma le HTTP request provengono dallo *stesso indirizzo IP*: comunque posso usare dispositivi (e browser) diversi...

Quindi, bisognerà "*inventare*" un nuovo modo per fare tutto ciò, siccome il protocollo HTTP non è sufficiente per farlo; infatti, si dice che HTTP è un protocollo *stateless*, ovvero le request HTTP non conservano nessun riferimento alle richieste precedenti fatte dallo stesso browser e device.

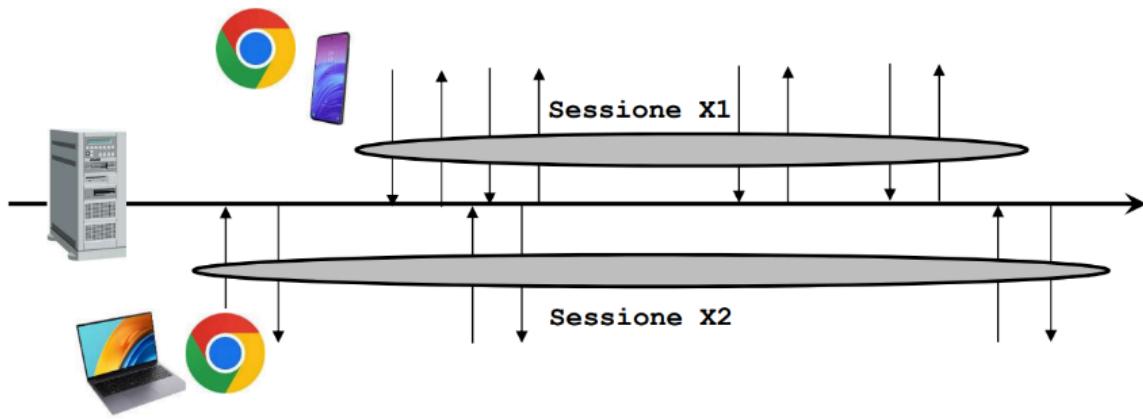
X

2. Sessione HTTP e Cookies

Definiamo il concetto della *sessione HTTP*:

DEFINIZIONE. (*Sessione*)

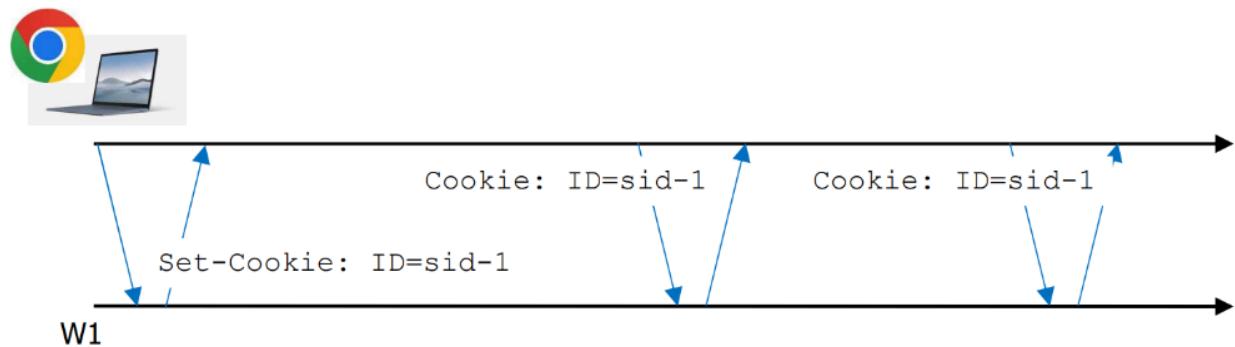
Una *sessione* è una sequenza di Request e Response HTTP scambiate da una coppia di *Browser@Device* e *Website*. Ogni sessione ha *identificatore univoco* sul Website.



In particolare, con l'implementazione mediante i **cookie** abbiamo che una HTTP request R appartiene ad una certa sessione **sid** con web server W *se e solo se* R contiene header **Cookie: name=sid**. Name e sid dipendono dal web server, sia dalla sua configurazione e dalla sua scelta fatta al momento.

Vediamo come si fa l'assegnazione di un cookie ad un **web browser**:

1. Web browser invia HTTP request senza header **Cookie**
2. Web server riceve la HTTP request, *crea* sessione identificata da **sid_1** e la registra nella sua **tabella delle sessioni**
3. Web server invia una HTTP response con header **Set-Cookie: ID=sid_1**
4. Web browser riceve la HTTP request, e salva il cookie nella propria **Cookie table**; nelle prossime HTTP request verso lo stesso host il WB inserisce il header **Cookie: ID=sid_1**



Approfondiamo sulle proprietà del **Cookie table**:

- Viene **condivisa tra tutti i tab**
- E' persistente, ossia fin quando non si decide di cancellarla rimane
- Non è condivisa tra browser diversi
- Può essere cancellata in qualunque momento

Q. Quanto possono durare i cookie?

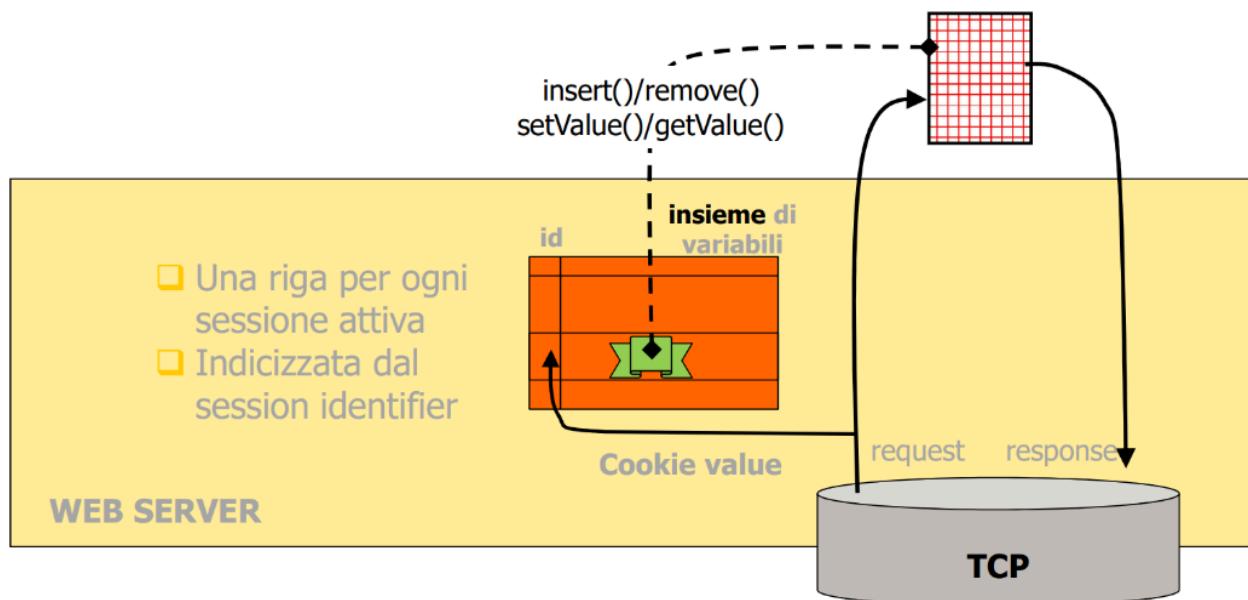
Può essere specificato dal web server, nel header **Set-Cookie: name=sid; Expires=date**

3. Stato delle Sessioni

Uno dei motivi per cui designiamo il concetto delle *sessioni* è proprio quello di associare delle *variabili* ad ogni sessione.

Una *variabile di sessione* è una *string*, che viene salvata in una *session state* gestita dal *web server*. Il web server dispone delle *funzioni* per agire sulla session state:

- *Creazione / Distruzione variabili*: `InsertSession("...", ...)`, `RemoveSession("...")`
- *Lettura / Scrittura variabili*: `SetValueSession("...", ...)`, `GetValueSession("...")`



Osservazione. Notiamo che abbiamo più aspetti delle variabili di sessione gestiti da livelli diversi:

- *Spazio di memoria e interfaccia di accesso*: Web server
- *Significato (interpretazione)*: Web app
- *Definizione header Cookie*: HTTP

X

4. Requisiti Cookie Identifier

Vedremo che ogni sessione può corrispondere ad un *acesso autenticato* ad una pagina web; quindi, se un giorno fosse possibile leggere la tabella dei cookie di qualcuno, le conseguenze sarebbero gravi! L'attacker può generare HTTP request che appartengono alle persone altrui.

Quindi il *cookie identifier* dovrà rispettare ulteriori requisiti, per essere più "*riservata*"

- *Non numerabile*: l'attacker non dev'essere in grado di poter "*indovinare*" la cookie id, sennò potrebbe tentare l'attacco eseguendo un ciclo while banale

- *Non predicibile*: non dev'esserci un pattern riconoscibile nella generazione delle session id

Authentication e Authorization

X

HTTP authentication. Concetti preliminari: assunzione dell'esistenza della sessione, definizione di subject e username, sessione autenticata e non autenticata. Protezione dei contenuti. Caso comune: Log-in con lo stesso URL. Azioni web server, pseudocodice. Concetto di authorization: diritto di accesso degli utenti, differenza dal concetto di authentication. Definizione di ACL (Access Control List), logica dell'authorization check. Generalizzazione degli ACL, definizione di Realm e Realm table. Osservazione: l'authorization va programmato in certi casi.

X

0. Voci correlate

- Sessioni HTTP

1. Concetto di Autenticazione

Supponiamo d'ora in poi l'esistenza della *sessione HTTP*.

Definizioni.

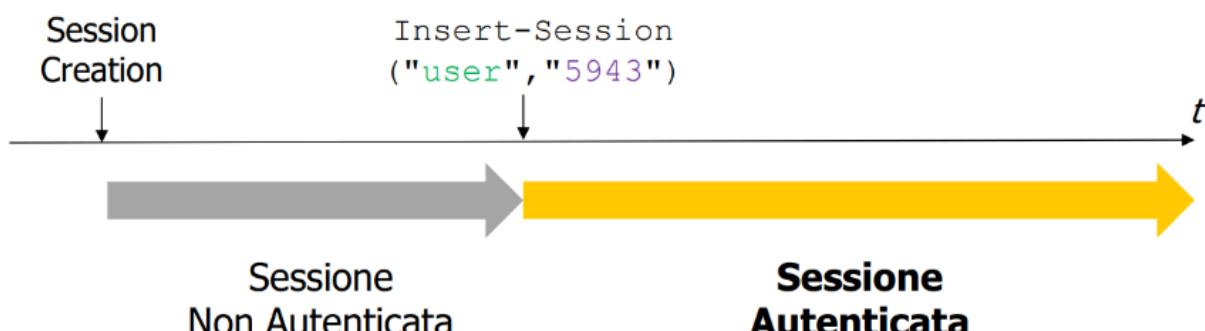
Subject: Utente di un sistema informatico (ESTERNO al calcolatore)

Username: Una stringa che identifica un'utente (INTERNO al calcolatore)

Un *subject* utilizza un *username* solo se è in grado di dimostrare di essere associato a tale username al sistema. Lo si fa mediante il possesso delle *credenziali*, e ci sono più modi per farlo; la più comune è la coppia Username-Password

Quindi ogni server si salva una *tabella delle credenziali*, dove ad ogni username si associa delle credenziali.

Definizione. Una sessione si dice *non autenticata* se non è associata a nessun username, altrimenti si dice *autenticata*. Ogni sessione è inizialmente *non autenticata* e può diventare *autenticata* mediante delle richieste HTTP. Di solito, la variabile di sessione **user** dipende dalla configurazione del web server.



Definizione. Un URL si dice protetto se richiede l'*autenticazione*, altrimenti si dice *non protetto*. Denotiamo gli URL protetti con URL-P e gli URL non protetti con URL-NP.

Per gestire il *log-in*, ci sono più casi:

1.1. Login Stessa Pagina

Analizziamo un caso comune: ovvero, se vogliamo accedere ad un URL-P di un sito web, dobbiamo effettuare il log-in (che si trova nello stesso URL-P)

Esempi: Facebook, Amazon

1. HTTP Request sessione non autenticata: **GET URL-P ...**
2. HTTP response con "*sollecita autenticazione*" (pagina log-in)
3. HTTP request con credenziali
4. HTTP response con documento dinamico personalizzato

1.2. Redirection verso Login Page

1. HTTP Request sessione non autenticata: **GET URL-P ...**
2. HTTP response con *3XX Redirection* verso **URL-LGIN**
3. HTTP request **GET URL-LGIN**
4. HTTP request con credenziali
5. HTTP response con documento dinamico personalizzato

Osservazione. In questo caso, come possiamo "*ricordarci*" a quale URL l'utente stava provando ad autenticarsi? Possiamo sfruttare la *session state* e salvare **URL-P** in una certa variabile della session state (che implicitamente assumiamo sia già presente, quindi il browser ha inviato header set-cookie eccetera...)

1.3. Login Diverso URL

1. HTTP Request sessione non autenticata: **GET URL-P ...**
2. HTTP response con "*sollecita autenticazione*" (pagina log-in)
3. HTTP request con credenziali **POST URL-VALIDATION**
4. HTTP reponse con *3XX Redirection* verso **URL-P1** predefinito
5. HTTP request **GET URL-P1**
6. HTTP response con documento dinamico personalizzato

Osservazione. Nei casi 1.2. e 1.3. che prevedono l'HTTP redirection, nella request **POST** in cui si trasmettono le credenziali come *Referer* si può inserire sia URL-P, URL-LGIN, ..., o anche ometterla siccome non è specificato in RFC

2. Protocolli di Autenticazione

Per mandare le richieste di "*solllecito autenticazione*" e di "*invio credenziali*", ci sono più modi. Vedremo i protocolli FORM e BASIC, le più comunemente usati

2.1. FORM

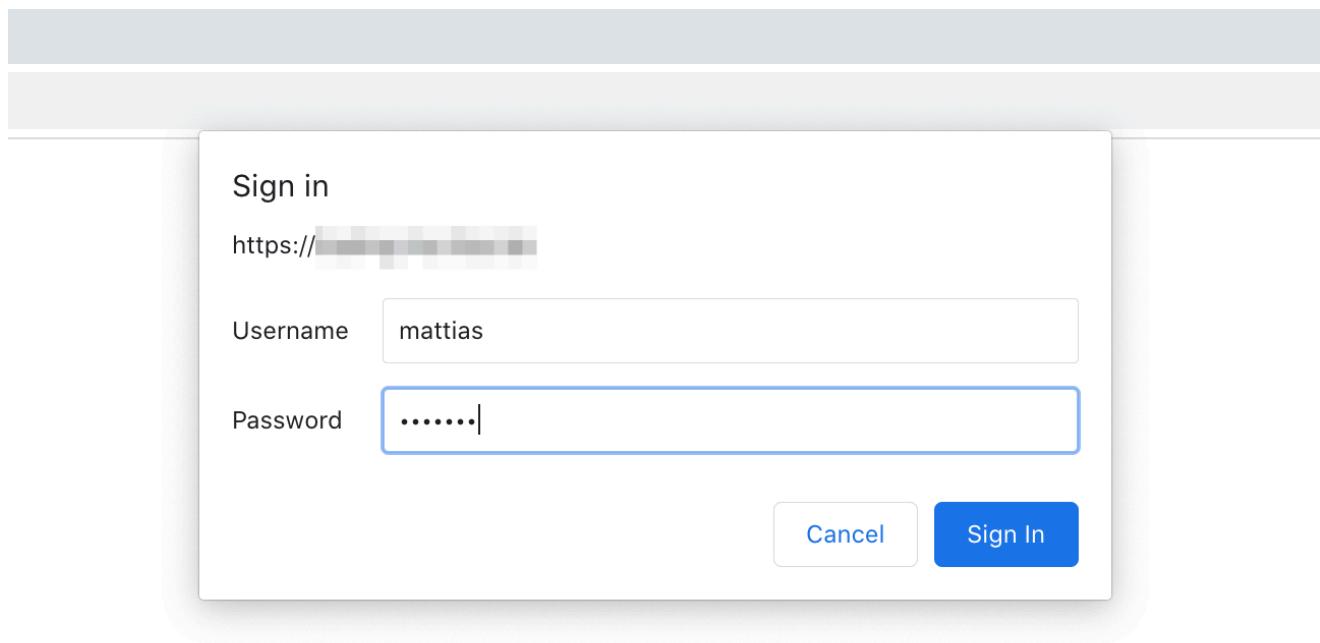
L'idea di base è semplicemente quello di richiedere le credenziali su un form HTML, in un *documento a parte*! (Ovvero non inserisco il FORM nel documento da proteggere, sennò sarebbe inutile...)

- *Solllecito*: 200 OK, inviare il documento in cui si sollecita ad inserire le credenziali
- *Invio Credenziali*: Request POST, con content-type "form" e verso un'URL di validazione

2.2. BASIC

L'idea di base è quella di aprire una *finestrella esterna alla pagina* per chiedere l'username e la password e inviare un'HTTP request specificando i dati in uno degli *header*

- *Solllecito*: 401 Unauthorized (!, faremo un'osservazione dopo), inserisco header **WWW-Authenticate: Basic realm = "<my_realm>"** e altre informazioni necessarie
- *Autorizzazione*: **GET URL_P** e metto header **Authorization: BASIC u:p** dove **u** e **p** sono codificate in BASIC64 (nota! non è una protezione vera e propria, in quanto è una codifica facilmente reversibile)



Notiamo che in nessun modo sono state usate delle funzionalità HTTP per l'autenticazione.

Inoltre, facciamo un'osservazione nel caso di BASIC authentication: quando il web server sollecita l'utente ad autenticarsi, si manda una response di codice 401 "Unauthorized" al client. Il termine *Unauthorized* è più che forvante, siccome in realtà si tratta di autenticazione e non c'entra niente con l'autorizzazione (vedremo dopo). D'altronde anche i termini usati negli header sono forvianti, in quanto si usa "*Authorization*" e "*Authentication*" intercambiabilmente.

3. Logica Web Server

Per fare tutto ciò, il server usa la seguente logica (in pseudocodice):

PYTHON

```
req_url <- extractURL(req)
if req_url in URL_NP:
    doc <- BuildDocument(req_url)
    Respond(doc)

elif req_url in URL_P:
    u <- getValue-Session("user")
    if not u:
        # sessione non autenticata
        user, pwd <- extractData(req)
        if user, pwd in DATA:
            doc <- BuildDocument(req_url, u)
            Respond(doc)

    else:
        Respond("You must login", ...)

else:
    doc <- BuildDocument(req_url, u)
    Respond(doc)
```

4. Authorization

Col concetto di *authentication* abbiamo distinto gli "*URL protetti*" da quelli non protetti. Questo basta? No, in quanto tutti gli utenti autenticati hanno gli stessi "*diritti d'accesso*".

Ad esempio, su Drive alcuni documenti sono visualizzabili *solo* da certi utenti. Come facciamo?

ACL. Prima di tutto, nella *configurazione del web server* definiamo gli *Access Control List*, che sono degli insiemi di utenti autorizzati a visionare la pagina dell'URL. Quindi, alla logica del *web-server* aggiungiamo un controllo aggiuntivo prima di restituire il documento: controllare che l'utente appartenga all'ACL dell'URL.

Tuttavia, è difficile definire individualmente gli ACL per ogni URL. Definiamo la seguente nozione per avere una rappresentazione più compatta degli ACL:

Realm. Un realm è un *insieme di risorse protette nello stesso modo*, formato da:

- *Realm Name*
- *Resources*: Descrizione molto compatta di risorse protette
- *ACL*: Insieme di username autorizzati
- *Protocol*: Vedremo dopo, si intende il protocollo di autenticazione (BASIC, FORM, eccetera...)

Realm Name	Resources	ACL	Protocol
Esami	/esami/*	alberto marco ruud	FORM
Documenti	*.pdf	alberto ricky andry	BASIC
Config	/admin/*	admin	FORM

In ogni realm è presente implicitamente la *Default Realm*, contiene tutti gli altri URL non protetti ed è accessibile senza autorizzazione.

Realm Name	Resources	ACL	Protocol
Esami	/esami/*	alberto marco ruud	FORM
Documenti	*.pdf	alberto ricky andry	BASIC
Config	/admin/*	admin	FORM
Default	Any other	-	No Auth.

Osservazione. Quindi con i *realm* l'authorization può essere un aspetto configurato nel *web server* in una tabella. Questo basta? No, ci sono certi casi in cui l'authorization va anche programmata. Esempio: dato un server web che usa un *programma* che esegue query a DB, bisogna stare attenti che l'utente sia effettivamente autorizzato ad effettuare la query.

Un caso notevole che esemplifica quanto sia necessario *programmare* i realm, è la vulnerabilità del sito web 18app: non è stato aggiunto il controllo di authorization nelle pagine individuali degli utenti, quindi ogni utente autenticato poteva accedere alle pagine (e consumare i voucher) di goni altro utente...

Q. Supponiamo che un browser@device sia autenticato ad un username. Cosa succede se cambio il *realm* durante la sessione? L'utente deve autenticarsi di nuovo?

Solitamente no, tuttavia se il nuovo protocollo di autenticazione è più "*forte*" allora si richiede di riautenticare.

X

5. Recap: Logica Web Server

Per fare un recap, si scrive un *pseudocodice* che descrive la logica del web server per gestire l'authentication e l'authorization (assumendo session).

```
let URL_P
let ACL(URL_1)
let ACL(URL_2)
...
let URL_NP

req <- GetRequest()
req_url <- ExtractUrl(req)

if req_url in URL_NP:
    doc = BuildDocument(req_url)
    SendResponse(..., doc)

elif req_url in URL_P:
    cookie <- req[cookie]
    if not cookie:
        # Creare sessione
        cookie = GenerateCookie()
        States[cookie] = {}
        SendResponse(..., header: {'set_cookie': f'id={cookie}'})

    elif cookie:
        usr = GetValueSession('user')
        if usr:
            # Se autenticato
            if usr in ACL(req_url):
                # Se autorizzato
                doc = BuildDocument(req_url, usr, ...)
                SendResponse(..., doc)
            elif usr not in ACL(req_url)
                SendResponse("Not Allowed", ...)

        elif not usr:
            # Se non autenticato
            if req['credentials']:
                # Se manda le credenziali
                u, p = req['credentials'].extract()
                if (u,p) in CREDENTIALS:
                    BuildDocument(u, doc, ...)
                    SendResponse(..., doc)

            elif not req['credentials']:
                SendResponse("Devi autenticarti", ...)
```


Modalità Incognito

X

Un paio di parole sulla navigazione in incognito. Definizione di modalità "incognito" di un browser.
Osservazioni pratiche.

X

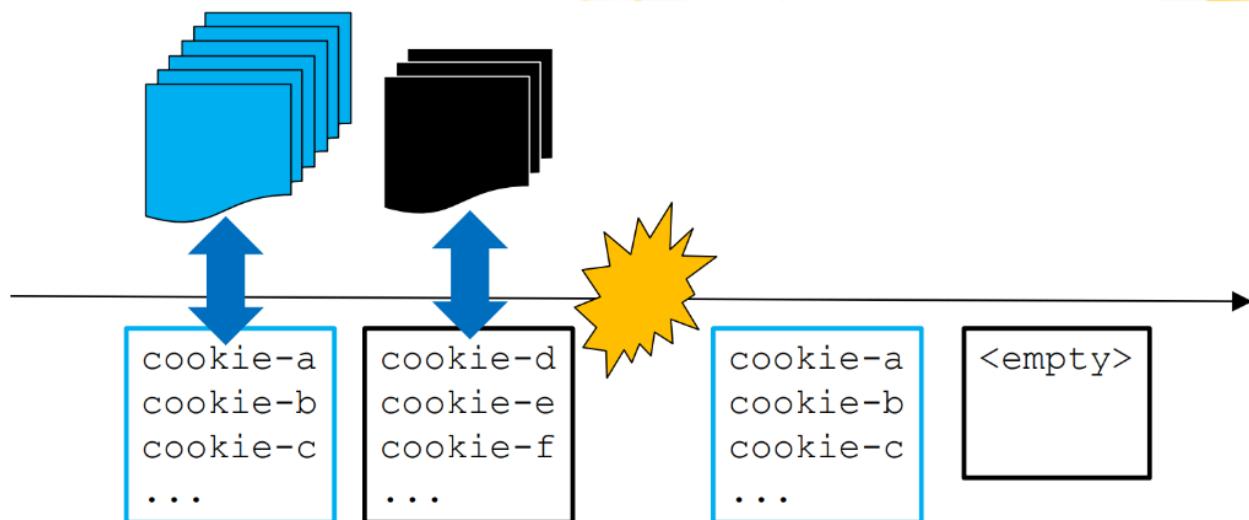
0. Voci correlate

- Sessioni HTTP

1. Modalità Incognito

Ricordiamo che una delle proprietà del *cookie table* è il fatto che sia *persistente* e *condivisa* tra i tab.

Definizione. La "*modalità incognito*" di un browser consiste in creare un'altra *cookie table separata* che non sia persistente. In altri termini, si tratta di un browser separato "*usa e getta*".



Esempi.

- Il browser naviga su un certo sito web www.somesite.it e la si accede come un certo utente U. Aprendo un altro tab e navigando sullo stesso sito, si rimane ad essere utente U; tuttavia, aprendo un tab incognito la si accede senza autenticazione
- Vale anche il viceversa, solo che la cookie table del browser incognito non persiste (rimane comunque tra i tab)

Q. Questo mi aiuta a navigare "*senza tracce?*"

Naturalmente no, siccome comunque ho molti fattori che "*lasciano tracce*":

- Comunicazioni DNS

- Associazioni a username
- ...

Quindi il termine *incognito* è un termine fuorviante, in quanto ci suggerisce l'idea di navigare anonimamente, anche se in realtà non stiamo facendo altro che usare un browser "*usa e getta separato*".

Per risolvere veramente il problema, si usano altre tecnologie (di cui non vedremo), come VPN o TOR browser. Inoltre, sarebbe definire *da chi* vogliamo nascondere le nostre tracce.

Web Analytics e Tracking

X

Aspetto pratico del web: web analytics e tracking. Problema di web analytics: contare accessi a siti, per pagina. Metodi per implementare web analytics. Tracking: cookie di terze parti, implicazioni..

X

0. Voci correlate

- Nozioni sul World Wide Web
- Pagina e Sito Web
- Sessioni HTTP

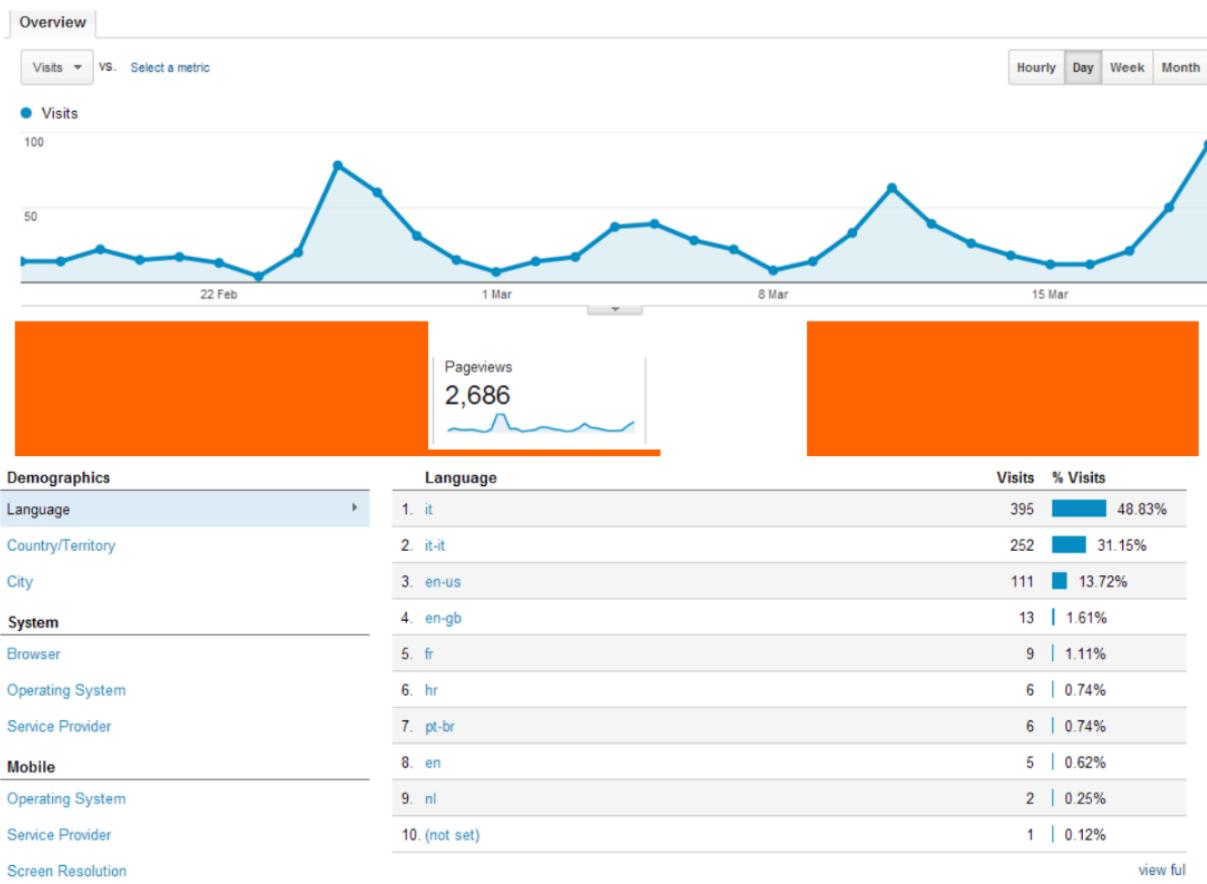
1. Web Analytics

Avendo tutti gli strumenti sul web, vediamo un paio di applicazioni.

PROBLEMA. Supponiamo di essere il proprietario di un sito. Come possiamo contare gli *accessi* al sito, dividendoli per *pagina*?

Il *conteggio* non sarà realizzato da noi, in quanto delegheremo questo lavoro ad un'altro web site A, noto come *servizio di web analytics* che è un *servizio specializzato* che facilita l'analisi dei dati.

Esempio. Un esempio noto di servizio analytics è *Google Analytics*. Non solo è presente delle informazioni sulle visite, ma anche gli *header delle HTTP request*, gli *indirizzi IP lato client*, eccetera...

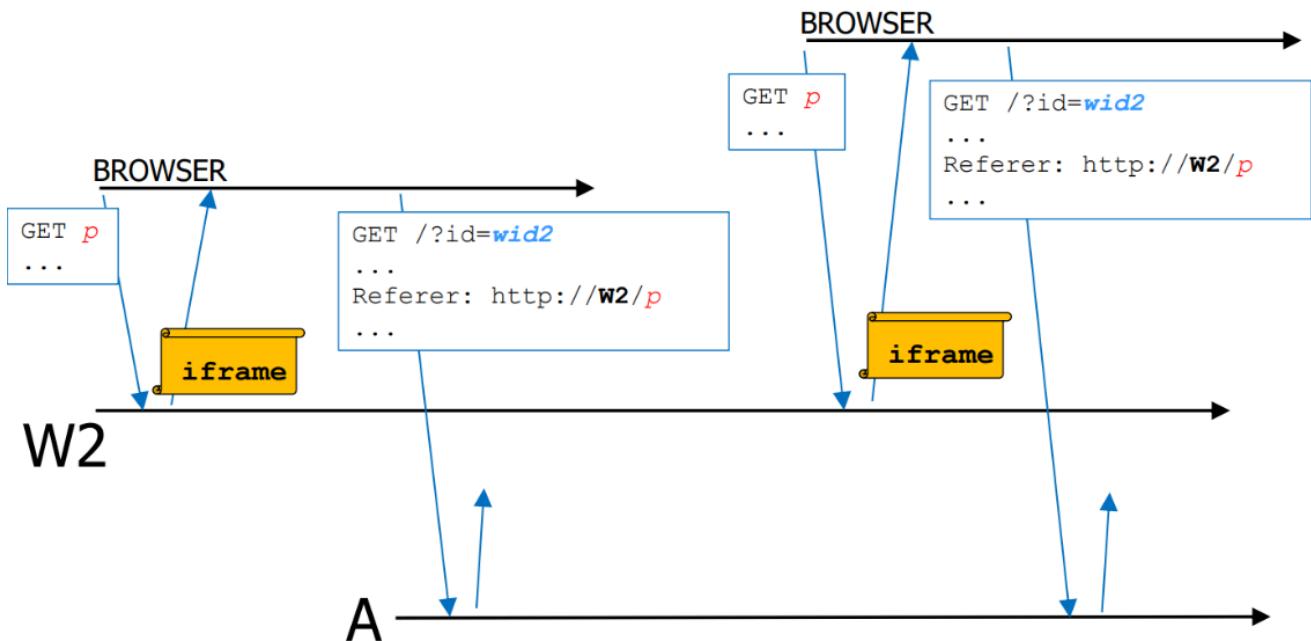


In pratica, per "*delegare*" questo lavoro al servizio A, ci sono tanti modi per farlo. L'unico requisito dev'essere che il *caricamento di una pagina P* implica automaticamente l'invio di una HTTP request ad A.

Esempio. A assegna un codice univoco (wid) ad ogni sito, e poi inseriamo un *iframe* "*invisibile*" (microscopicamente piccolo) in ogni pagina: `<iframe src="http://A/?id=wid" height=1 width=1>`

Notiamo che in questo esempio:

- Il codice WID è ridondante, siccome nel referer specificheremo l'URL "*sorgente*". Tuttavia, è più utile nel senso pratico per semplificare le implementazioni su database (in quanto le WID formeranno una chiave primaria eccetera)
- A conterà *solo le visite*, e non è in grado di identificare l'utente siccome non ha delle informazioni sufficienti. Con *tracking* vedremo che sarà possibile farlo lo stesso (con grossissime implicazioni...)



Esempi. Altri esempi:

- Stesso di prima ma con "immagini trasparenti o molto piccole"
- Script JS
- Sistemi di sviluppo leb, vengono inserite automaticamente
-

Osservazione. Ogni pagina web W può anche usare più servizi di web analytics A1, ..., An.

X

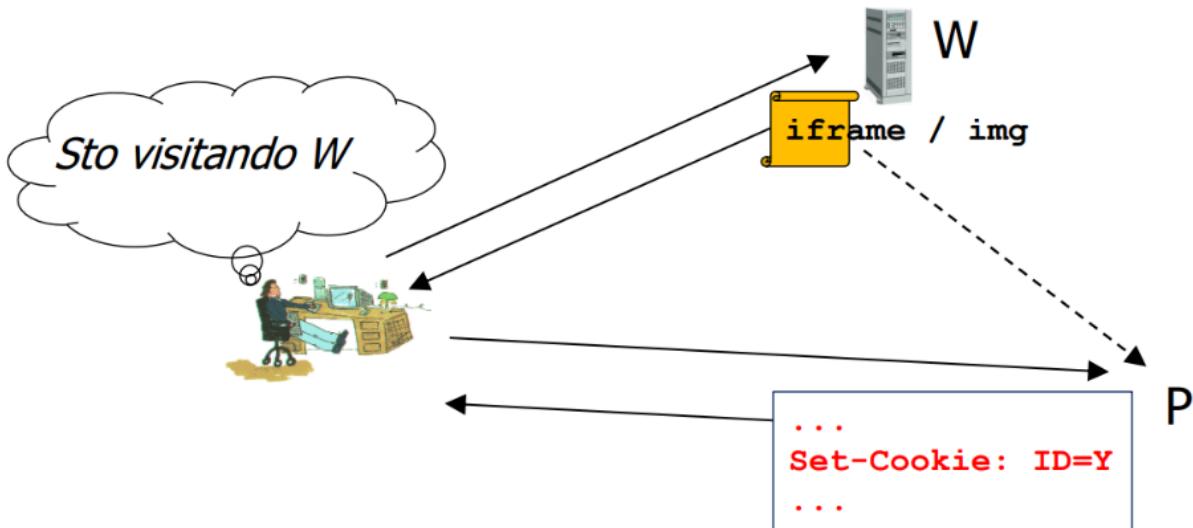
2. Third-Party Cookies

Andiamo ad *aumentare* l'obiettivo di prima:

PROBLEMA. Raccogliere *sequenze di pagine visitate da ogni browser* e sapere *il numero di clienti "abituale"*

La soluzione a questo problema consiste nell'usare i *cookies* per gli accessi ad A, con una scadenza "*quasi infinita*". Notiamo che con questa soluzione saremo anche in grado di *contare i browser!*

THIRD-PARTY COOKIES. Notiamo che l'utente *crede di visitare* il sito web W, ma in realtà anche P e P può creare un cookie da assegnare all'utente; questo cookie si dice *cookie di terze parti*. Questo scenario è presente anche in contesti più generali, oltre al web analytics.



Motivazioni. Le motivazioni per usare i *third-party cookies* sono le seguenti:

- *Web Analytics*: come visto prima
- *Tracking*: in questo caso P non aiuterà W a fornire dei dati, invece W permette a P di raccogliere i dati (in cambio di denaro) da raccogliere come suoi, e li venderà ad altri organizzazioni che vogliono creare *profili* di Browser@Device

Il secondo caso è *enormemente diffuso*, ogni W ha tipicamente molti P.

Implicazione. Notiamo subito l'implicazione dei *third-party cookies*: se un sito P è *"presente su molti siti web W"*, allora potrebbe in grado di riconoscere *"ogni"* Browser@Device. Ciò vuol dire quindi P sarà in grado di osservare la navigazione di *"ogni"* Browser@Device...

Per mitigare questa situazione, l'*informativa GDPR* obbliga ai siti W europei di *informare di ogni third-party cookie* a cui forzano gli utenti. In particolare, vengono specificate che le finalità devono essere *"chiare"* e il consenso deve essere *esplicito e consapevole*.

Web Advertising

X

Web advertising (cenni di cenni): idea di base del funzionamento, ruolo di Ad Network. Modi di fare web advertising: contextual advertising e targeted advertising. Implicazione di targeted advertising.

X

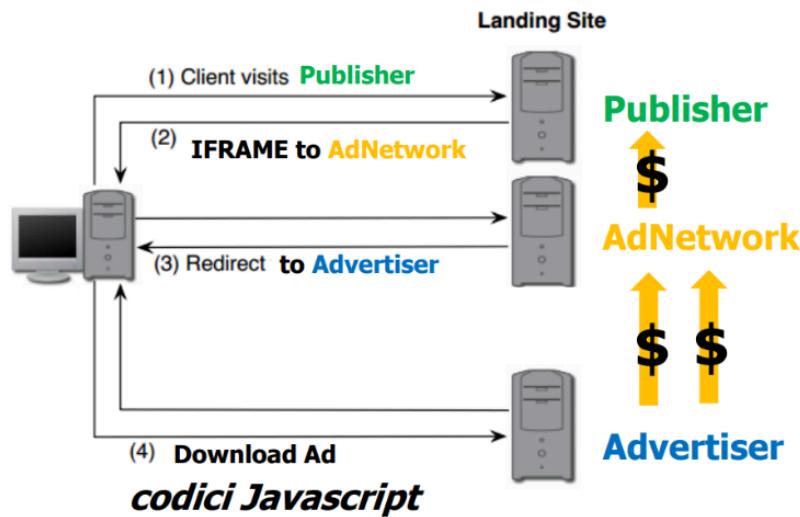
0. Voci correlate

- Web Analytics e Tracking
- Pagina e Sito Web

1. Web Advertising

Il *web advertising* consiste nello seguente schema:

1. Il client visita il sito web "*publisher*" P, che fornisce i contenuti e uno *spazio pubblicitario*. P si dice anche "*Landing site*"
2. Lo spazio pubblicitario non è altro che un tag *iframe* che andrà a prelevare la risorsa dal web server "*Ad Network*" N. L'*ad network* non è altro che l'intermediario tra gli *advertiser* e le *landing site*
3. Contattando N, si viene reindirizzati all'*advertiser* che in risposta invia la pubblicità effettiva, mediante dei codici JS



Il motivo per cui si fa questo schema sono *soldi*, in particolare si paga l'intermediario N per *click* e *per view*. In particolare, i click sono più "*profittabili*" di un'ordine di grandezza rispetto alle view; ciò incentiva gli advertiser A a creare *pubblicità interessanti*.

Osservazione. In questo schema, Ad Network fa delle scelte "*on the spot*" per decidere verso quale *advertiser* indirizzare. Ciò si basa su *algoritmi e infrastrutture complicate* che rispettino i

seguenti requisiti:

- Veloce e scalabile
- Ottimo (massimizza la "*probabilità*" di ottenere click)
- Mostrare tutti gli advertiser

Quindi ci saranno due modi di *scegliere* l'advertiser:

1. *Contextual*: Tengo conto solo del sito corrente come contesto, ossia scelgo l'advertiser più "*adatta*" alla pagina in cui viene visualizzata. L'algoritmo si basa quindi sull'analisi del contenuto di P e delle pubblicità
2. *Targeted*: Qui invece ci basiamo sul *profilo dell'utente* a cui si mostra la pubblicità. Ciò porta con se delle *implicazioni economiche, sociali, politiche e strategiche enormi*, dal momento in cui si cercherà di raccogliere più informazioni sul *Device owner* possibile per avere un targeting ottimale.

Concludiamo dicendo che questo è il *vero costo* dei "*servizi apparentemente gratuiti*" sul web: al posto di pagare con i soldi, paghiamo con i nostri dati che vengono raccolti.

"Mail"

Architettura Email

X

Architettura e-mail. Terminologia: definizione di mail, e-mail. Architettura dell'infrastruttura mail: definizione di Mail User Agent, Mail Transfer Agent, dominio Mail Server. Connessioni e protocolli (cenni): tipologie usate. Scenario comune di sistemi mail. Esempio completo: sender to receiver.

X

1. Significato di "e-Mail"

Nel gergo tecnico, il termine *e-mail* (o semplicemente *mail*) può assumere più accezioni:

- L'indirizzo mail, come ad esempio **john@gmail.com**
- Il messaggio contenuto nella mail
- L'infrastruttura mail di cui vedremo

Per noi, il termine *mail* andrà a significare il *messaggio stesso*, ovvero una sequenza di linee di testo come specificato da un RFC. Per ora, sappiamo che nell'email si ha gli indirizzi mail del mittente e del destinatario.

X

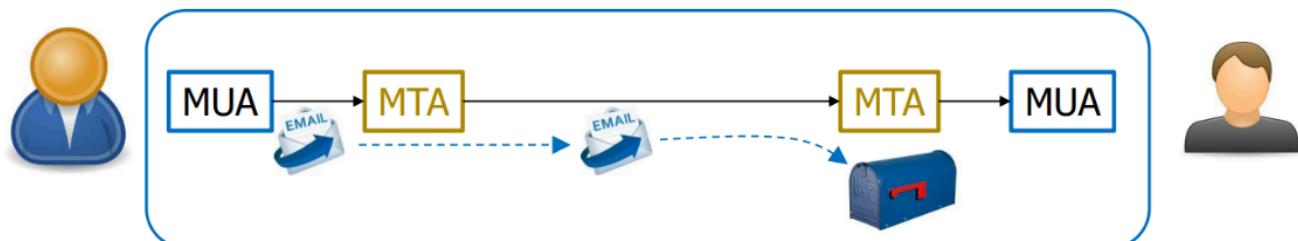
2. Architettura Mail

L'architettura mail è composta da due sottosistemi, e sono le seguenti:

DEFINIZIONE. Le *mail user agent* (MUA) sono i programmi utilizzati dagli utenti per gestire la propria email; quindi invio, ricezione, scrittura, archiviazione ed eccetera...

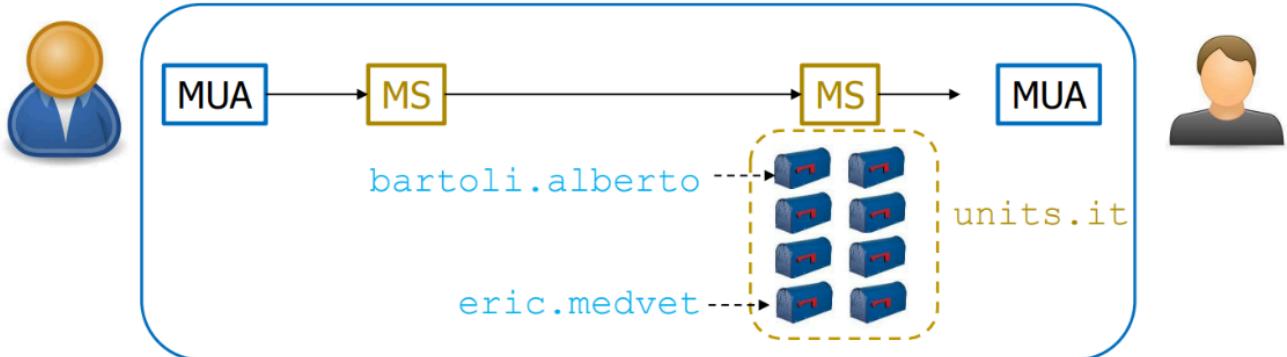
Una MUA può essere quindi un *programma installato sul computer* o una *web app* (web mail).

DEFINIZIONE. Le *mail transfer agent* sono invece dei *programmi server* che tipicamente forniscono due servizi: il trasporto delle mail e la memorizzazione degli email ricevuti (*mailbox*). Quindi, in approssimazione possiamo dire che sono delle *mail server* (MS).



Quindi una MUA interagisce col "suo" MTS (circa MS) per inviare delle mail.

Proprietà Mailbox. Ogni *mailbox* è identificata (e anche localizzata) *univocamente* dal suo *indirizzo email*. La sua sintassi è intuitivamente formata da due parti, divisa dal simbolo chiocciola @. La prima parte è l'identificatore della mailbox, la seconda è il *dominio e-mail*; un *dominio e-mail* è un insieme di mailbox e si trovano tutti sullo stesso MS.



Q. Dato un dominio e-mail, come possiamo rintracciare il suo (o uno dei suoi) mail server?

Semplice, basta interagire col DNS effettuando richieste su RR di tipo MX. Osserviamo che quindi le *mailbox* non centrano niente con gli RR!

Osservazione. Notiamo che tutte le mailbox di un dominio si devono trovare sullo stesso MS, ma è possibile che un dominio e-mail abbia più mail server. Esempio: vedere con la propria posta elettronica istituzionale, usando *Dig Online* per consultare gli RR

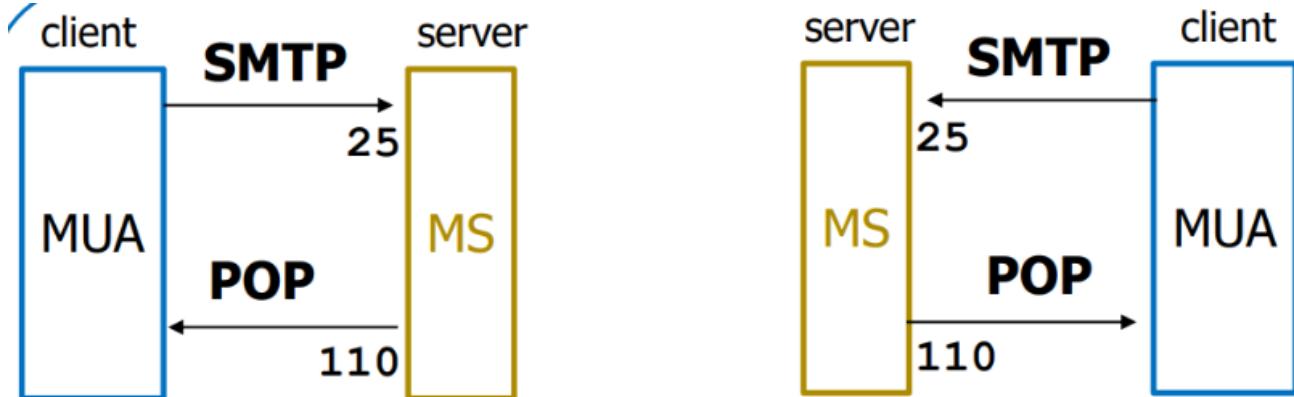
Osservazione. Dominio e-mail e mail server possono avere nomi completamente diversi tra loro.

X

3. Tipologie Connessioni

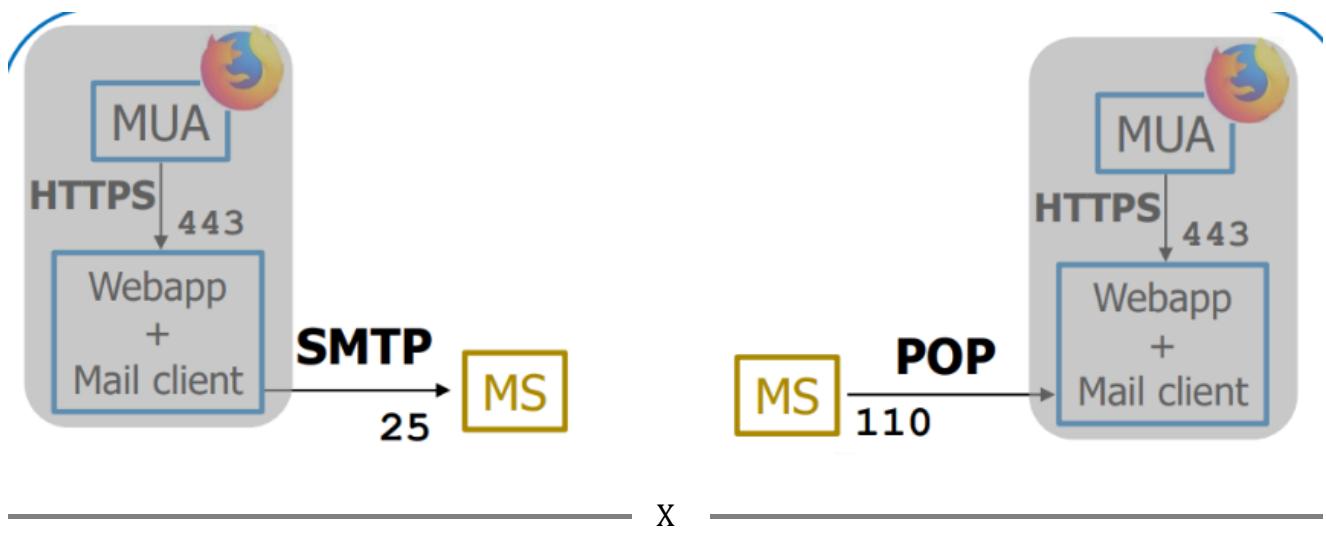
Accenniamo su come avviene la *comunicazione* in termini di *connessioni TCP e protocolli*.

1. *MS to MS per inviare le mail*: Si usa il protocollo *SMTP*, porta 25 lato server. Qui non c'è *mai* il bisogno di autenticarsi, altrimenti sarebbe da distribuire una tabella delle credenziali in *tutti i mail server* che sarebbe assurdamente impossibile da implementare
2. *MUA to MS (invio mail)*: Per inviare le mail, si usa il protocollo *SMTP* con porta 25 lato MS. Qui l'autenticazione è "*opzionale*", nel senso che come descritti nelle RFC non c'è bisogno di autenticarsi, ma praticamente oggi è quasi sempre necessario autenticarsi. Questo aspetto è problematico e si riconduce al caso dell'*e-mail spoofing* (vedremo dopo cosa vuol dire)
3. *MUA to MS (estrazione mail)*: Per consultare le mail ricevute, si usa il protocollo *POP* con porta 110 lato MS. Qui è sempre necessaria l'*autenticazione*.



Osservazione. Nel caso 2. notiamo che la MS non fa mai niente di propria spontanea volontà; è sempre la MUA a contattare la MS, anche nel caso in cui riceviamo mail.

Osservazione. Nel caso 2., se abbiamo che la MUA è implementata come *web app*, allora essa non ha connessione diretta con MS. Invece essa contatterà la *web app* e la *mail client* mediante protocollo HTTP, che a sua volta contatterà la MS.

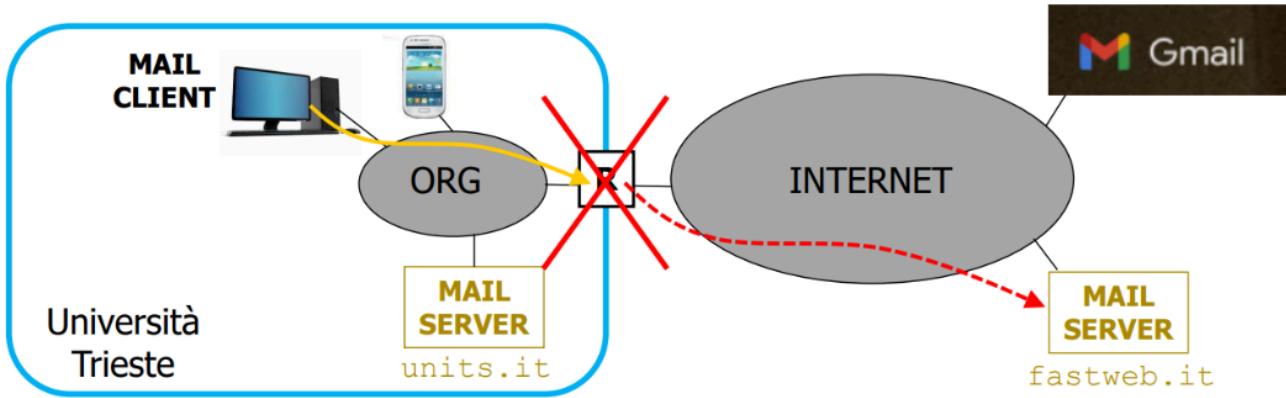


3. Configurazione Sistema Mail

Un scenario *comune* delle organizzazioni è quello di configurare il traffico nella seguente maniera:

- Non permettere traffico e-mail diversi da "*quello proprio*", ovvero il router di frontiera R bloccherà qualsiasi connessione di tipo SMTP verso un mail server esterno che non provenga dal proprio mail server interno.

Lo si fa per motivi pratici, per controllare "*cosa entra e cosa esce*".



Notiamo che in questo caso è possibile comunque usare Gmail o un'altra web mail, siccome la MUA si connette al mail client esterno mediante HTTP/HTTPs.

4. Sender to Receiver

Vediamo l'esempio completo: come *"viaggia"* un'email, quando viene inviato?

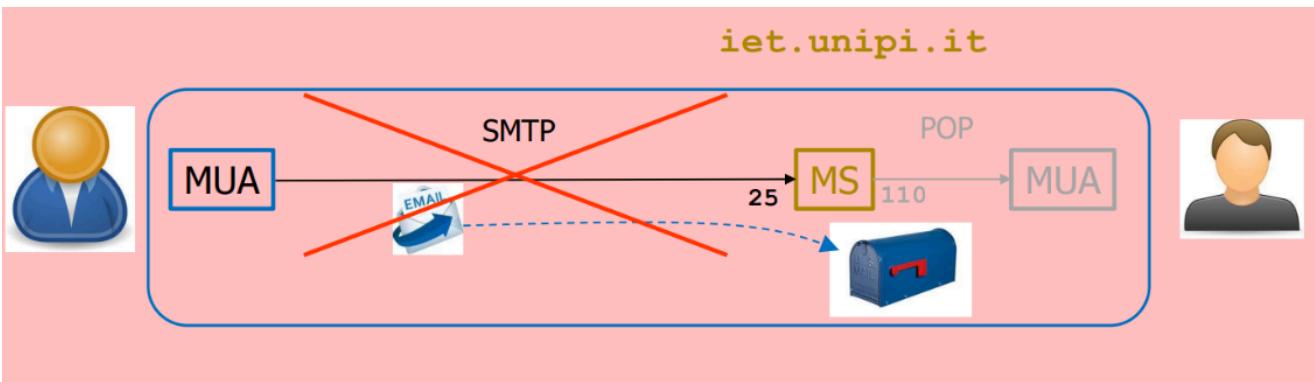
1. Partiamo dalla *mail client*, ovvero il programma che invierà richiese al MS. Essa dev'essere configurata:

- Da avere un *indirizzo e-mail*
- Tipicamente ha anche una *username + password* (di solito username è il nome della mailbox)
- Ogni mail client può quindi gestire più *indirizzi email*, per ora assumiamo che ne gestisca solo uno per semplicità concettuale

2. Supponiamo che adesso il mail client sia pronto per inviare la mail al destinatario. Abbiamo due casi:

1. Il *destinatario* ha lo stesso dominio e-mail del *mittente*: in questo caso è già *"arrivato"*, se lo deposita e basta
2. Se sono *diversi*, allora bisognerà invece fare richieste al DNS per determinare l'indirizzo dell'altro Mail Server (MX e poi A, oppure solo MX se il DNS dispone delle RR aggiuntive)

Osservazione. In questo paradigma il *mail client* si collega *soltamente* al proprio Mail Server, e *mai* ad altri *mail server* indipendentemente da dove si trova l'utente! Anche (e soprattutto) per trasmettere mail!



Questa era una cosa possibile tempo fa, ma non si fa più (a meno che non si vuole "*aggirare*" il sistema per qualche motivo strano...)

Protocolli e-Mail. Protocollo POP: definizione, paradigma e notazione. Comandi da sapere. SMTP: definizione, osservazioni su SMTP: versione autenticata e/o criptata. Comandi SMTP secondo RFC 821, formato delle mail secondo RFC 822.

0. Voci correlate

- Architettura e-Mail

1. Protocollo POP (RFC 1225)

Il protocollo **POP** (*Post Office Control*) è un protocollo a *comandi/risposte* ed è *text-based* su ASCII. Ovvero, i comandi e le risposte vengono codificati in caratteri ASCII. Una risposta viene codificata con:

- **+OK** se il comando inserito ha un esito positivo
- **-ERR** altrimenti, con informazione aggiuntiva

Una risposta può essere contenuta su più linee, e viene terminata col carattere punto **.**

NOTAZIONE. Per descrivere la comunicazione fatta mediante POP, denotiamo con **S** i comandi inviati dal server e **C** quelli dal client. In realtà non esiste, lo facciamo solo per chiarezza visiva

COMANDI. Vediamo i comandi inviabili dal lato client:

- **USER** **user@example.com**, **PASSWORD** **password**: Autenticarsi alla mailbox propria
- **LIST**: Elencare i messaggi. Il lato server mostrerà un elenco numerato di mail ricevute, con size dei messaggi relativi
- **RETR** **n**: Leggere la mail **n**-esima di quella ottenuta nell'elenco
- **DELETE** **n**: Eliminare la mail **n**-esima di quella ottenuta nell'elenco
- **QUIT**: Terminare la comunicazione

Esempio. (*Comunicazione tipica POP*)

```
S: +OK POP3 SERVER READY
C: USER alice@example.com
S: +OK USER ACCEPTED
C: PASS secret123
S: +OK PASSWORD ACCEPTED
C: LIST
S: +OK 2 MESSAGES
S: 1 1200
S: 2 850
S: .
C: RETR 1
S: +OK 1200 OCTETS
S: <mail>
S: .
C: DELE 1
S: +OK MESSAGE 1 DELETED
C: QUIT
S: +OK GOODBYE
```

Osservazione. In questo caso, è il *lato server* che inizia la conversazione: prima di poter procedere, deve dare "l'OK" al client.

2. Protocollo SMTP (RFC 821)

SMTP è un protocollo *molto intricato*, per motivi storici. Ricordiamo che ci sono due "*step*" in cui si va a comunicare col protocollo SMTP:

1. *MUA to MS*: Qui l'autenticazione e/o criptaggio è "*opzionale*". All'inizio non c'era proprio
2. *MS to MS*: Qui non c'è l'autenticazione

Quindi ci sono più "*versioni*" del protocollo SMTP che sono state aggiunte nel tempo, per poter aggiungere la possibilità di autenticazione e/o criptaggio. In totale, ci sono *3 numeri di porta diversi* con funzionalità auth/crypto diverse.

Nel corso *trascuriamo* l'uso della crittografia, e faremo finta di usare solo la porta 25.

SMTP. Similmente a *POP*, *SMTP* è un protocollo *ASCII text-based* a *comando/risposta*. Da un punto di vista funzionale, nelle risposte conta solo l'*identificativo numerico* (anche se c'è in realtà del testo descrittivo dell'esito, per aiutare l'utente).

COMANDI. Vediamo i comandi inviabili dal lato client:

- **EHLO/HELO <mail_domain>**: Il client si identifica come il *mail server* di un certo mail domain.
- **MAIL FROM: <s@sender.com>**: Iniziare a mandare una mail, dichiarando dapprima il mittente
- **RCTP TO: <r@receiver.com>**: Specificare il destinatario
- **DATA**: Dichiarare che si sta iniziando a comporre la mail.
- **QUIT**: Terminare la comunicazione
- **AUTH LOGIN**: Dichiarare che si sta iniziando ad autenticarsi. Da qui la mail server potrà chiedere l'username e la password, che verranno inseriti in *codifica Base64*

Vediamo un esempio tipico:

```

S: 220 mail.receiver.com ESMTP Postfix
C: EHLO mail.sender.com
S: 250-mail.receiver.com
S: 250-AUTH LOGIN PLAIN
C: AUTH LOGIN
S: 334 ...
C: ...
S: 334 ...
S: 235 2.7.0 AUTHENTICATION SUCCESSFUL
C: MAIL FROM:<s@sender.com>
S: 250 Ok
C: RCPT TO:<r@receiver.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: Subject: Server-to-server test
C: This is a test message sent without authentication.
C: .
S: 250 Ok: queued as 67890

```

X

3. Formato Mail (RFC 822)

Q. Con SMTP e POP possiamo vedere le *mail*. Come si scrive in effetti una mail?

Osserviamo innanzitutto che per *mail* - inteso come il *messaggio* - non si ha solo il contenuto, ma l'*involturlo in sé*. Ovvero, essa è composta da due parti:

Headers: Insieme di header lines **Name: Value**, in cui si hanno delle informazioni aggiuntive per gestire le mail.

Hanno un ruolo analogo alle headers HTTP; tuttavia, la differenza consiste nel fatto che in questo caso gli header possono essere generati da più processi:

- *Mail Client mittente*
- *Mail Server intermedi*
- *Mail Client ricevente*

Osserviamo che di solito nelle applicazioni MUA vediamo solo un *sottoinsieme degli header*, di solito c'è l'opzione per vederli tutti

Body: Il contenuto vero e proprio della mail. Essi sono codificati in *linee di caratteri ASCII 7-bit*, e l'ottavo bit dev'essere sempre impostato a 0 (per ragioni storiche e intricate di cui non conosciamo).

Per allegare dei file o per avere più bit, abbiamo delle *estensioni optional* da parte della MUA sender per trasformare il body originale in ASCII 7-bit, e poi nell'header si specificano *"come"* vengono effettuate queste trasformazioni; così la MUA receiver sarà in grado di effettuare la trasformata inversa del body.

Email Spam e Address Spoofing

X

Problemi pratici legati ai protocolli email. Spam e Address spoofing. Mitigazioni allo spam e address spoofing.

X

0. Voci correlate

- Protocolli Email

1. Problemi Pratici

Ci sono dei *problemi pratici molto importanti* legati all'architettura mail e ai protocolli usati. Questi problemi non hanno delle soluzioni in quanto sono intrinseche al modo in cui è stata designata l'architettura mail e configurazione delle mail server, bensì abbiamo delle *mitigazioni abbastanza "efficienti"*.

I problemi sono due:

Spam:

- Invio massimo di email "*indesiderati*"
- Fini commerciali e spesso illegali
- Oggi compongono la maggioranza degli email inviati

Spoofing:

- Falsificazione indirizzo dell'email del mittente.
- Questo è "*facilissimo*" da fare! Infatti, osserviamo che nel protocollo SMTP sarebbe teoricamente possibile specificare il mittente di una mail senza autenticazione. Per farlo, basterebbe collegarsi direttamente ad una mail server e inviare una mail specificando l'indirizzo mittente falsificato... (usando i comandi **MAIL FROM** e l'header **From: ...**)
- In realtà oggi è molto difficile da fare, siccome le "*mitigazioni*" che abbiamo oggi sono piuttosto efficaci

In pratica le mitigazioni ai due problemi consistono in applicare delle *euristiche* per conoscere mail *spam o "spoofed"*. Come, ad esempio:

- Indirizzi SMTP diversi da indirizzi in email
- Domini dichiarati in EHLO non esistenti
- Header mancanti
- Analisi del testo usando algoritmi ML (nota! oggi questi algoritmi hanno un'accuracy del circa 0.99)
- Uso di protocolli ancora più specifici per l'autenticazione

- eccetera...

Ma i problemi persistono perché:

- **Spam**: Le mail spam sono moltissime, e quindi nonostante l'accuratezza dei modelli di classificazione, alcuni vengono comunque "fatte passare"
- **Spoofing**: L'attacker, se disposto di risorse sufficienti, sarà comunque in grado di falsificare una mail che passi i filtri euristici descritti. Inoltre, un'alternativa più semplice del **spoofing** è quello semplicemente di **ingannare** registrando domini mail con nomi che "**assomiglino**" a certi enti (principio derivante dal caso delle DNS). Esempio: l'attacker può comprare il dominio **polizia-postale-it.it** e crearcì delle RR di tipo MX..

Collegamento Diretto con i Server Internet

X

Cuoriosità pratica: programmi per aprire connessioni a server manualmente. Programma telnet, esempio con email (SMTP e POP). Problemi pratici dell'esempio.

X

0. Voci correlate

- Comunicazione tra Processi
- Proprietà dei Servizi di Comunicazione
- Protocolli Email

1. Connessione Diretta con i Server Internet

RICHIAMO. Ricordiamo che ogni *server* implementa un protocollo, che descrive come devono essere formattate le request o response. Molti protocolli hanno request/response *a linee di caratteri*.

Allora si potrebbe comunicare "*direttamente*" con i server che implementano tali protocolli dal proprio PC, usando il programma *telnet*. Tramite il programma posso *aprire connessioni a server* e *ciò che scrivo è il messaggio*, e il *messaggio che vedo stampato a schermo è la risposta*.

I comandi per usare telnet sono semplicemente le seguenti:

- **telnet**: Aprire il programma
- **open <server_name> <portnumber>**: Aprire una connessione
- **set LOCAL_ECHO**: Per vedere ciò che viene stampato a schermo

Osservazione. Il programma funziona in un modo tale che i backspace non vengano riconosciuti come caratteri, ovvero è impossibile "*correggere*" il testo. L'unica cosa che si può fare è quello di interire enter e sperare che ci sia uno "errore di sintassi"

X

2. Esempio di Utilizzo: Interazione coi Mail Server

Esempio. Ad esempio, si può usare telnet per interagire coi *mail server*. Tuttavia, ci sono moltissimi problemi che renderebbe questo caso d'utilizzo difficile, tra cui:

- Il router di frontiera potrebbe proibirci il traffico SMTP
- Il Mail Server potrebbe chiudere la connessione in quanto non ci riconosce come un altro mail serve
- Il Mail Server potrebbe chiudere la connessione in quanto non siamo autenticati

Quindi ci sono altri programmi che ci permettono di fare tutto ciò, tra cui:

- **swaks** per specificare tutte le opzioni nelle SMTP request
- **openssl s_client** per criptare
- PuTTy per gestire protocolli criptati

Aspetti Vari delle Mailbox

X

Aspetti vari sulle mailbox: *forwarding*, *mailing list* e *send as*.

X

0. Voci correlate

- Architettura e-Mail

1. Aspetti Misti

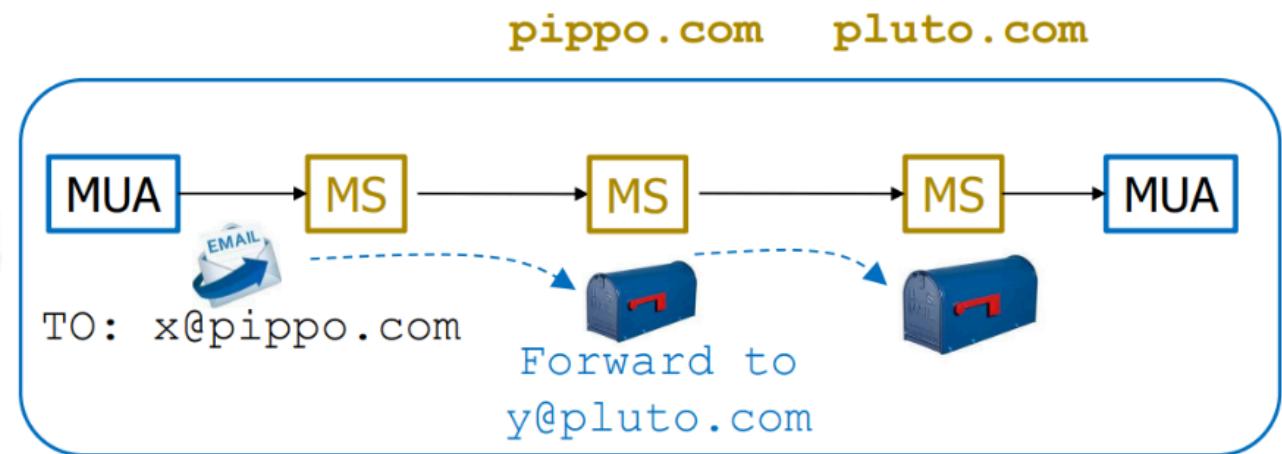
Supponiamo di avere una *mail* che "passa" tra due mail server, MS e MS'. Se $MS \neq MS'$, allora si dovrebbe aspettare che la mail *passi* tra solo *due server*.

1.1. Mail Forwarding

In realtà vedremo che non è sempre vero, per gli aspetti sulle mailbox che vedremo.

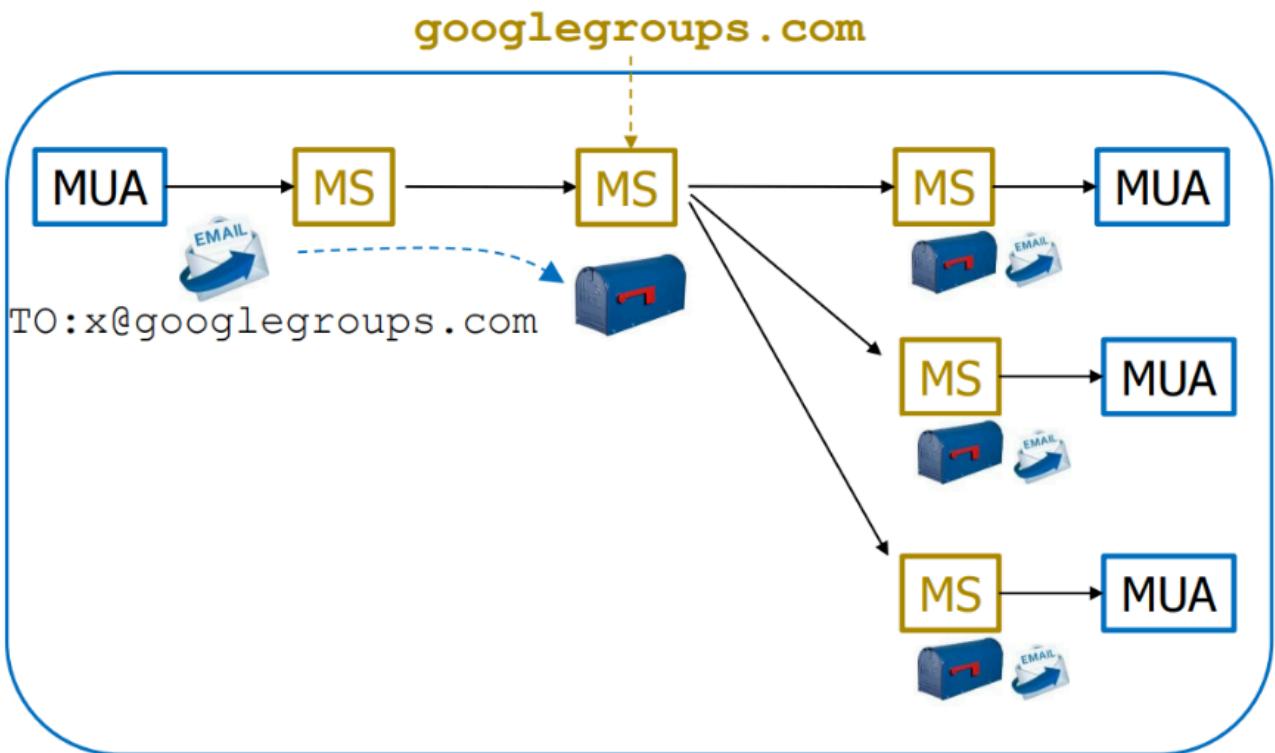
Cominciamo dapprima con la *forwarding*: in pratica, ogni *mailbox-A* può essere configurata per *inoltrare* una mail ricevuta verso ad un'altra *mailbox-B*, e optionalmente cancellare la mail.

Quindi, in questo caso si ha che la mail passa per *tre server*.



1.2. Mailing List

In altri *mail server*, è possibile configurare una *mailbox* come una *mailing list*; ovvero di associare un *insieme di mailbox* ad una *mailing list*, e ogni email ricevuto dalla mailing list viene trasmessa a *tutte le mailbox* nella mailing list.



Osservazione. La mailing list può comprendere mailboxes di *dominio diverso!* Quindi la mail passa per più mail server

1.3. Send As

In alcuni *mail client* (come *gmail*), è possibile scegliere di mandare le mail con un'*identità diversa*. Tuttavia, ciò non vuol dire che posso usare un dominio email *mittente* arbitrario! (sennò sarebbe spoofing...)

In realtà funziona che il *mail server* invia la mail all'altra *mail server* (che si riferisce alla "*identità scelta*"), tramite protocollo SMTP *con autenticazione*; dopodiché il mail server lo invia come di consueto.

Quindi tutto ciò che serve per collegarsi all'altro mail server è definito *in configurazione*, tra cui username, password, eccetera...

Come con la forwarding, abbiamo che la mail passa per *tre* mail server.

