Il livello applicativo dello stack ISO/OSI

A cura del prof. Giuseppe Mastrandrea
Sistemi e Reti

A.S. 2019/2020 I.I.S. A. Righi - Cerignola (FG)

INDICE

INDICE **2**

Introduzione **4**

Comunicazione fra processi **5**

Architetture delle applicazioni di rete **7**

Protocolli di livello applicativo **8**

Status Code **10**

Posta elettronica **11**

SMTP 12

Formato di un messaggio di posta elettronica 14

Content-Type 15

Return-Path 16

Accesso alla mailbox 16

POP3 17

Configurazione di un client di posta 19

FTP **22**

HTTP **25**

Architettura 27

Risorse 28

Formato della richiesta[7] 29

Formato della risposta 31

Un esempio di interazione HTTP 32

Strumenti per sviluppatori del browser 32

Comandi HTTP testuali 35

Persistenza dello stato in HTTP 36

Cookie 37

Gli header Set-Cookie e Cookie 37

Evoluzione del concetto di risorsa in HTTP 40

Bibliografia/Sitografia **41**

Introduzione

Le applicazioni di rete sono la ragione di essere delle reti di calcolatori. Se non riuscissimo a concepire applicazioni utili non ci sarebbe alcun bisogno di progettare protocolli di rete per supportare le applicazioni. Sin dagli esordi in Internet sono state sviluppate numerose applicazioni utili. Tali applicazioni sono state la forza propulsiva del successo di Internet, spingendo le persone a fare in modo che la rete fosse parte integrante della loro vita quotidiana a casa, sul lavoro, nelle scuole e negli uffici pubblici.

Sono applicazioni per Internet quelle che divennero popolari negli anni '70 e '80, tra cui **e-mail**, **trasferimento file** e **newsgroup**, ma anche la *killer application* di metà degli anni '90: il **World Wide Web**, che comprende la navigazione sul Web, la ricerca di informazioni e il commercio elettronico (e-commerce). Altre due applicazioni per Internet sono la **messaggistica istantanea** e la **condivisione di file** tramite P2P: le due killer application introdotte a fine millennio. Dall'anno 2000 abbiamo assistito a un'esplosione di applicazioni popolari voce e video tra cui la telefonia su IP (voice-over-IP, **VoIP**), **la videoconferenza su IP** come Skype, lo **streaming di video** generati dagli utenti come YouTube e il **cinema on demand** come Netflix, senza contare le piattaforme che ci consentono di ascoltare **musica in streaming**. E ancora **giochi on-line multiutente**, come Second Life e World of Warcraft Più recentemente sono comparse le applicazioni di **social networking** di nuova generazione come Facebook e Twitter, che hanno creato una rete di persone sopra la rete Internet composta da router e collegamenti fisici. E il processo di creazione di nuove, eccitanti applicazioni per Internet non si arresta: magari sarà qualcuno di voi a creare la prossima killer application di Internet!

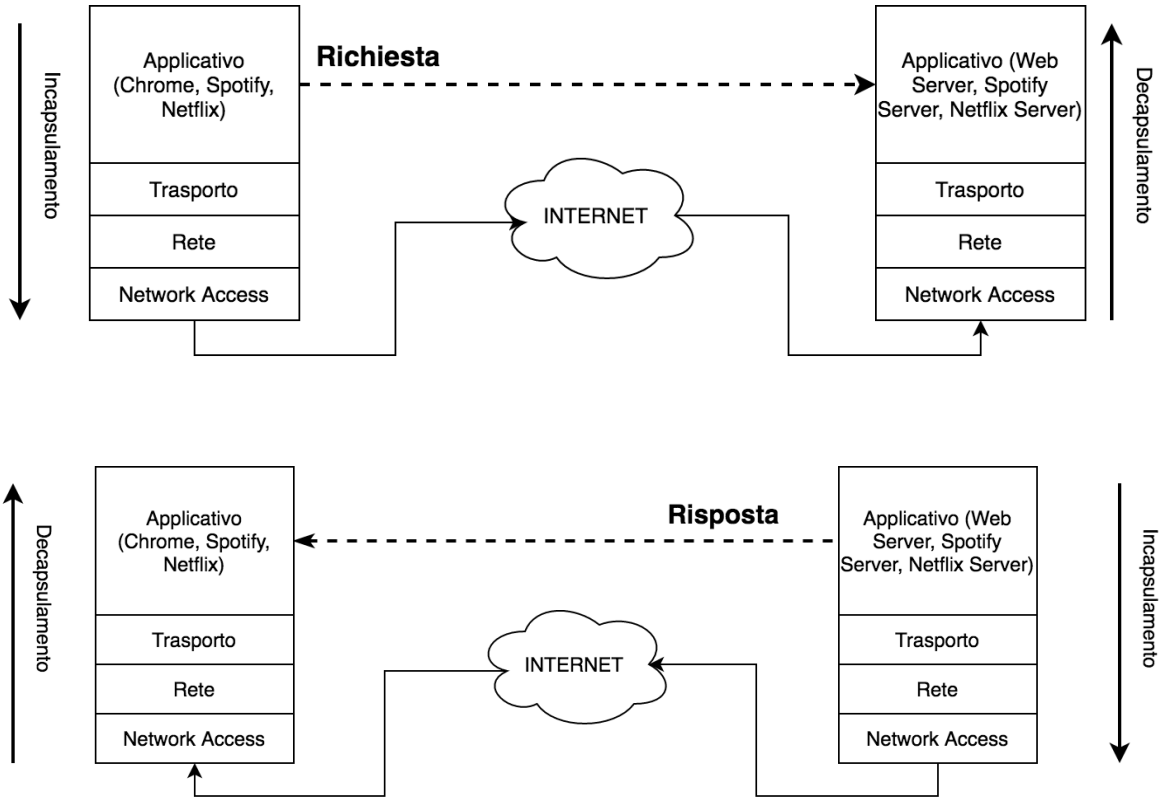
Killer application significa letteralmente "*applicazione assassina*", ma viene intesa in gergo nel senso metaforico di un'applicazione decisiva, vincente, rivoluzionaria o di estremo successo, grazie alla quale la tecnologia su cui essa si basa (Internet nel nostro caso) riesce a penetrare il mercato condizionando in maniera anche importante il modo di vivere delle persone.

In queste dispense, una rielaborazione del libro “Reti di calcolatori e internet. Un approccio top-down” di James F. Kurose e Keith W. Ross affronteremo gli aspetti concettuali delle applicazioni di rete. Inizieremo definendo i concetti chiave del livello di applicazione le architetture dei servizi di rete. Di seguito esamineremo in dettaglio alcune applicazioni, tra cui Web, posta elettronica, condivisione di file.

Il cuore dello sviluppo delle applicazioni di rete è costituito dai programmi che sono eseguiti dai sistemi periferici e che comunicano tra loro via rete. Per esempio, nelle applicazioni web esistono due programmi diversi che comunicano tra di loro: il **browser**, che viene eseguito dall'host dell'utente (desktop, laptop, smartphone, etc) e il **web server** che si trova nell'host che viene di solito chiamato “server”. Considerando un altro esempio, in un sistema di condivisione di file tramite P2P (Peer to Peer) troviamo un programma su ogni host che partecipa alla comunità di condivisione. In questo caso, i programmi in esecuzione sugli host possono essere simili o identici.

Comunicazione fra processi

Prima di costruire un' applicazione di rete dovrete anche conoscere come comunicano tra loro i programmi in esecuzione su diversi sistemi terminali. Nel gergo dei sistemi operativi non si parla in effetti di programmi, ma di processi comunicanti. Si può pensare a un processo come a un programma in esecuzione su un sistema. Processi in esecuzione sullo stesso sistema comunicano utilizzando un approccio interprocesso (interprocess communication). Le regole di questo tipo di comunicazione sono governate dal sistema operativo del calcolatore in questione. Ma in questa fase non siamo interessati a come comunicano i processi all'interno dello stesso host, bensì a come comunicano i processi in esecuzione su sistemi diversi, che potrebbero anche avere sistemi operativi diversi. I processi su due sistemi terminali comunicano scambiandosi messaggi attraverso la rete: il processo mittente crea e invia messaggi nella rete e il processo destinatario li riceve e, quando previsto, invia messaggi di risposta.



Nelle figure precedenti un esempio di interazione request/response. Il processo client (un browser web, l'app di Spotify o di Netflix) manda dei messaggi di richiesta ad un altro processo server (un web server, o il server di streaming di netflix o il server di streaming di spotify), e ci dà la parvenza (indicata dalla linea tratteggiata) di essere connessi direttamente al processo remoto. In realtà il messaggio viene incapsulato in fase di invio in messaggi via via più grandi,

viaggia sulla rete Internet e lato ricevente viene decapsulato fino ad arrivare al messaggio di livello applicativo originariamente inviato.

Le applicazioni di rete sono quindi costituite da una coppia di processi che si scambiano messaggi su una rete. Per esempio, nelle applicazioni web, un browser (processo client) scambia messaggi con un web server (processo server). In un sistema di condivisione P2P, i file sono trasferiti dal processo di un peer al processo in un altro peer. Per ciascuna coppia di processi comunicanti, generalmente ne etichettiamo uno come client e l'altro come server. Nel Web, il browser rappresenta un processo client, mentre il web server è un processo server. Nella condivisione di file tramite P2P il peer che scarica il file viene detto client mentre quello che lo invia è chiamato server. In alcune applicazioni, come quelle di condivisione dei file via P2P, un processo può essere sia client sia server, in quanto può tanto inviare quanto ricevere file. Ciò nondimeno, nel contesto di una data sessione tra una coppia di processi, possiamo ancora etichettare l'uno come client e l'altro come server. Definiamo come segue i processi client e server:

Nel contesto di una sessione di comunicazione tra una coppia di processi quello che avvia la comunicazione (cioè, contatta l'altro processo all'inizio della sessione) è indicato come ***client*** *mentre quello che attende di essere contattato per iniziare la sessione è detto server.*

Nel Web, ad esempio un processo browser avvia il contatto con un processo web server; quindi il primo è il client e il secondo il server. Nella condivisione di file P2P, quando il peer A chiede al peer B di inviare un dato file, il primo rappresenta il client e il secondo il server, nel contesto di questa specifica sessione di comunicazione. Occasionalmente, useremo anche la terminologia "lato client e lato server di un'applicazione".

Ad ogni programma è associato un numero, il cosiddetto **numero di porta**, che viene utilizzato per tenere traccia del mittente e del ricevente di un messaggio di livello applicativo. I numeri di porta vanno da 0 a 65535 (occupano infatti 16 bit). I primi 1024 (da 0 a 1023) numeri sono detti anche **well-known ports**, e sono i numeri associati solitamente ai processi server. L’associazione fra i numeri di porta e i processi client, invece, è affidata al sistema operativo, il quale sceglie numeri arbitrari compresi fra 1024 e 65535. La gestione dei numeri di porta è affidata al livello di trasporto.

L’oggetto software tramite cui è possibile tenere traccia di una connessione fra due applicazioni è il **socket**. Esso può essere di due tipi:

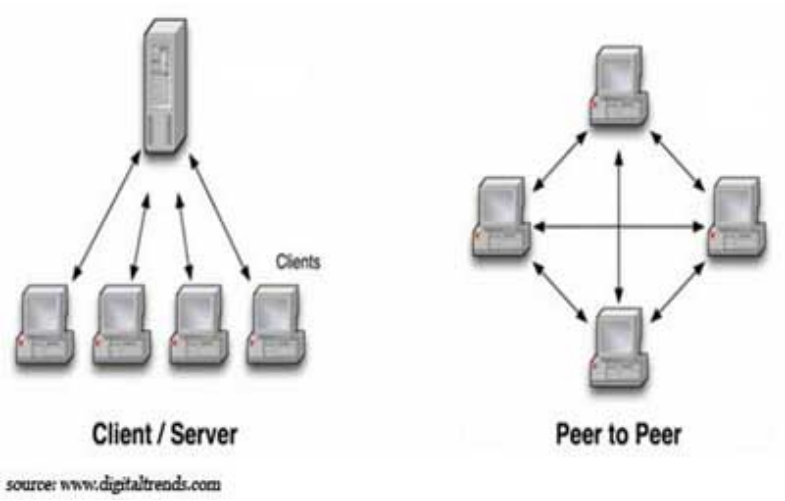
- Socket UDP: formati solo da una coppia IP/numero di porta
- Socket TCP: formati da 2 coppie IP/numero di porta per un totale di 4 campi

Architetture delle applicazioni di rete

Prima di occuparci del software vero e proprio è necessario disporre di un progetto dettagliato dell'**architettura dell'applicazione**. Essa descrive come devono interagire i programmi che implementano una particolare applicazione di rete. Nella scelta lo sviluppatore si baserà probabilmente su una delle due principali architetture attualmente utilizzate:

- Architettura client-server
- Architettura P2P.

Nell'architettura client-server vi è un host sempre attivo, chiamato *server*, che risponde alle richieste di servizio di molti altri host, detti *client*. Un esempio classico è rappresentato dall' applicazione web, in cui un web server, sempre attivo, risponde alle richieste dei browser in funzione sui client. Quando riceve una richiesta di una risorsa da parte di un client, il server risponde con la risorsa richiesta. Si osservi che nell'architettura client-server i client non comunicano direttamente tra loro; allo stesso modo, in un'applicazione web, i browser non interagiscono direttamente tra loro. Inoltre il server dispone di un indirizzo fisso (IP o nome host da convertire con DNS). Il client può quindi contattare il server in qualsiasi momento, inviandogli un messaggio. Tra le più note applicazioni con architettura client-server, ricordiamo il Web, il trasferimento dei file con FTP, Telnet e la posta elettronica.



In un'architettura P2P l'infrastruttura di server è minima o del tutto assente; si sfrutta, invece, la comunicazione diretta tra coppie arbitrarie di host, chiamati *peer* (ossia pari), collegati in modo intermittente. I peer non appartengono a un fornitore di servizi, ma sono dispositivi come computer, smartphone, etc, controllati dagli utenti, che per la maggior parte si trovano nelle abitazioni, nelle università e negli uffici. Dato che i peer comunicano senza passare attraverso un server specializzato, l'architettura viene detta peer-to-peer. Molte tra le applicazioni attualmente più diffuse e con elevata intensità di traffico sono basate su un'architettura P2P. I servizi di rete che funzionano su P2P includono la condivisione di file (per

esempio, BitTorrent), la telefonia su Internet (per esempio, Skype) e IPTV (per esempio, Kankan e PPstream). Abbiamo accennato che alcune applicazioni presentano un'architettura ibrida, combinando sia elementi client-server sia P2P. Per esempio, per molte applicazioni di messaggistica istantanea, i server sono usati per tenere traccia degli indirizzi IP degli utenti, ma i messaggi tra utenti sono inviati direttamente tra gli host degli utenti, senza passare attraverso server intermedi.

Protocolli di livello applicativo

Abbiamo visto come i processi di rete comunichino tra loro inviando messaggi tra socket. Ma come sono strutturati questi messaggi? Qual è il significato dei loro campi? Quando vengono inviati? Queste domande ci conducono nel campo dei protocolli a livello di applicazione. Un protocollo a livello di applicazione definisce come i processi di un'applicazione, in esecuzione su sistemi periferici diversi, si scambiano i messaggi. In particolare, un protocollo a livello di applicazione definisce:

- tipi di messaggi scambiati (per esempio, di richiesta o di risposta)
- la sintassi dei vari tipi di messaggio (per esempio, quali sono i campi nel messaggio e come vengono descritti)
- la semantica dei campi, ossia il significato delle informazioni che contengono
- le regole per determinare quando e come un processo invia e risponde ai messaggi.

Alcuni protocolli a livello di applicazione vengono specificati nelle **RFC** e sono pertanto di pubblico dominio. Per esempio, il protocollo a livello di applicazione del Web, **HTTP** (hypertext transfer protocol), è descritto nell' **RFC 2616**. Se lo sviluppatore di un browser si attiene alle regole che vi sono esposte, il suo browser sarà in grado di recuperare pagine web da qualsiasi server che segua quelle stesse regole. Altri protocolli a livello di applicazione sono privati e volutamente non disponibili al pubblico (per esempio Skype).
È importante distinguere tra applicazioni di rete e protocolli a livello di applicazione. Un protocollo a livello di applicazione è solo una parte (benché molto importante) di un'applicazione di rete. Un protocollo di livello applicativo **implementa** una particolare applicazione di rete. Consideriamo un paio di esempi. Il Web è un'applicazione client-server che consente agli utenti di ottenere su richiesta documenti dai web server. L'applicazione web consiste di molte componenti, tra cui uno standard per i formati di documento (HTML), browser (per esempio: Firefox e Google Chrome), web server (per esempio: Apache, Microsoft IIS) e un protocollo a livello di applicazione. HTTP, il protocollo a livello di applicazione del Web, definisce il formato e la sequenza dei messaggi scambiati tra browser e web server. È, pertanto, solo una parte dell'applicazione web (benché importante). Come ulteriore esempio consideriamo l'applicazione di **posta elettronica** su Internet. Anch'essa presenta molte componenti tra cui: server di posta che ospitano le caselle degli utenti, client di posta (come Microsoft Outlook) che consentono agli utenti di leggere o creare messaggi, protocolli a livello di applicazione che definiscono le modalità di scambio dei messaggi tra server diversi, tra server e client e come interpretare le

intestazioni dei messaggi. Di conseguenza, SMTP (simple mail transfer protocol) [RFC 5321], il principale protocollo a livello di applicazione per la posta elettronica, rappresenta solo una parte (sebbene consistente) dell'applicazione di posta elettronica.

Una tipica sessione di un protocollo di livello applicativo è costituita semplicemente dai messaggi di livello applicativo che vengono scambiati fra il programma che fa da client e il programma che fa da server. A seconda delle caratteristiche di affidabilità e velocità richieste dall'applicazione di rete, il protocollo può "poggiarsi" su TCP o su UDP. Se un'applicazione di rete dovrà essere più **affidabile** che veloce, userà quasi sicuramente **TCP** come protocollo a livello di trasporto; se un protocollo invece deve privilegiare la **velocità** sull'affidabilità avrà molto più senso utilizzare **UDP** come protocollo a livello di trasporto.

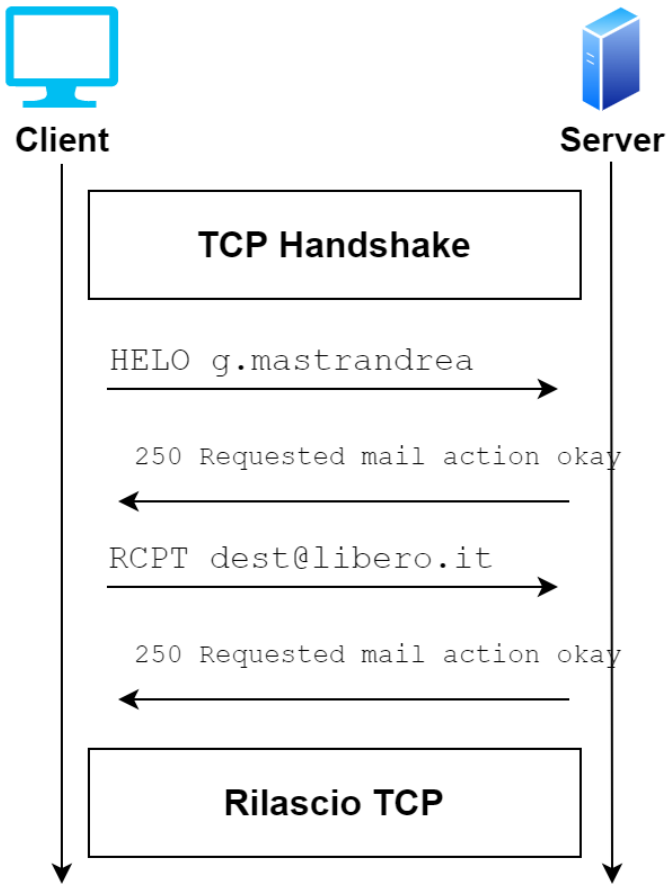
I messaggi che vengono scambiati fra programma client e programma server sono perlopiù in formato testuale, il che significa che virtualmente possiamo utilizzare tutti i servizi di livello applicativo utilizzando la riga di comando. In generale, una richiesta è formata da:

- **Header** (opzionali): sono una serie di coppie chiave/valore che servono ad arricchire la richiesta con ulteriori informazioni
- **Nome** del comando: rappresenta la vera e propria richiesta che viene effettuata dal client
- **Argomenti** del comando: rappresenta i dati che vengono mandati a corredo del comando
- **Payload** del comando (opzionale): rappresenta i dati aggiuntivi ed eventuali che vengono passati al server

Analogamente, il formato di una risposta nella stragrande maggioranza dei casi è di questo tipo:

- **Header** (opzionali): come nelle richieste, sono una serie di coppie chiave/valore che servono ad arricchire la risposta con ulteriori informazioni
- **Status code**: è un numero a 3 cifre che ci dà informazioni sullo stato della richiesta (vedi par. successivo)
- **Messaggio informativo**: è un messaggio correlato allo status code che l'utente può visualizzare accanto allo status code
- **Payload** della risposta (opzionale): è la risposta vera e propria, ad esempio

Di seguito un esempio di interazione fra un client ed un server nel protocollo SMTP. Notiamo che SMTP è un protocollo per la posta elettronica, che è un'applicazione di rete affidabile. Privilegiando l'affidabilità, il protcollo si poggia su TCP. Questo significa che prima dell'inizio della sessione SMTP (cioè prima dello scambio delle richieste/risposte SMTP) ci sarà bisogno di instaurare la connessione, e una volta terminato lo scambio di messaggi avverrà il rilascio della connessione:



Nella precedente sessione SMTP, ad esempio, le richieste non hanno header nè payload. Il comando inviato dal client è “HELO”, l’argomento del comando è “g.mastrandrea”. La risposta invece è costituita dai soli campi “Status Code” (250) e dal messaggio informativo (“Requested mail action okay”).

Status Code

Gli **status code** sono dei codici numerici che indicano se una specifica richiesta è stata completata con successo. Il meccanismo è il seguente:

1. Un client spedisce una richiesta ad un server (web, di posta elettronica, FTP, etc)
2. Il server riceve la richiesta e la elabora
3. In base al risultato dell’elaborazione, il server prepara una risposta e la re-invia al client
4. All’interno della risposta creata dal server c’è lo status code, dalla cui analisi possiamo dedurre se la richiesta è andata a buon fine

Uno status code quindi è spedito insieme ad una **risposta** dal server e si riferisce allo stato di una **richiesta** del client. Gli status code sono formati da 3 cifre, e si raggruppano in 5 classi a seconda del valore della prima cifra:

- **1YZ**: indica una risposta di tipo esclusivamente informativo
- **2YZ**: indica che la corrispondente richiesta è andata a buon fine

- **3YZ**: indica che la corrispondente richiesta è andata parzialmente a buon fine o è andata a buon fine ma c'è bisogno di altri dati per portarla a termine
- **4YZ**: indica che la corrispondente richiesta **non** è andata a buon fine per errori lato client o che la richiesta è stata temporaneamente rifiutata
- **5YZ**: indica che la corrispondente richiesta **non** è andata a buon fine per errori lato server o che la richiesta è stata definitivamente rifiutata

Di seguito alcuni esempi.
In SMTP[2]:

- Uno status code pari a 250 significa "Richiesta completata con successo"
- Uno status code pari a 354 significa "Richiesta ricevuta, continua ad inserire dati"
- Uno status code pari a 452 significa "Impossibile completare la richiesta: spazio insufficiente su disco"
- Uno status code pari a 500 significa "Errore di sintassi, comando non riconosciuto"
- Uno status code pari a 550 significa "Errore durante l'invio, mailbox non disponibile"

In HTTP[3]:

- Uno status code pari a 200 significa "Richiesta completata con successo"
- Uno status code pari a 301 significa "La risorsa desiderata si trova ad un altro URL"
- Uno status code pari a 404 significa "Risorsa non trovata"
- Uno status code pari a 503 significa "Servizio non disponibile"

Posta elettronica

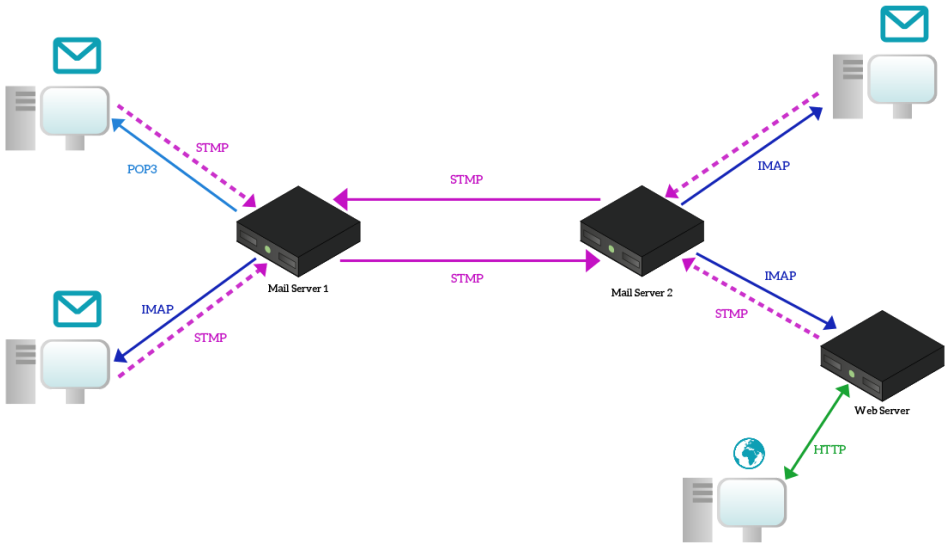
[1]La posta elettronica è un ottimo modo per comunicare sulla rete Internet. Le mail consentono lo scambio di contenuto testuale puro o *rich* (vale a dire arricchito con immagini, video, contenuti multimediali). Il contenuto è consegnato pressochè istantaneamente al destinatario corretto, e in caso contrario il mittente è informato di un qualsiasi errore avvenuto durante la consegna. A differenza del web, che funziona grazie ad un solo protocollo, la posta elettronica ha bisogno di almeno due protocolli per funzionare correttamente, visto che i processi in uno scambio di messaggi di posta elettronica sono almeno due: **spedire email** e **accedere alla casella di posta** (d'ora in avanti chiamata **mailbox**)**.**

Contrariamente a quanto si possa pensare di primo acchito, il servizio "Posta elettronica" funziona con protocolli client/server. In particolare, distinguiamo due famiglie di protocolli, ciascuna dedicata ad un compito esatto:

- Protocolli di **PUSH** (cioè i protocolli che servono ad inviare dei dati): usati per il solo **invio di posta elettronica**; di questa famiglia fa parte **SMTP** (Simple Mail Transfer Protocol)
- Protocolli di **PULL** (cioè i protocolli che servono ad ottenere dei dati): usati per il solo **accesso e la fruizione** della mailbox; di questa famiglia fanno parte i protocolli **POP3** (Post Office Protocol 3) e **IMAP** (Internet Message Access Protocol)

Tutti i sistemi di posta elettronica fanno uso di server. Un server mail è semplicemente un software che è in grado di ricevere messaggi e conservarli in una mailbox e di far accedere alle proprie mailbox gli utenti proprietari delle mailbox stesse.

A grandi linee, il processo di scambio di messaggi di posta elettronica funziona in questo modo: ogni volta che spediamo una mail, il nostro client di posta elettronica comunica con il SMTP del nostro provider di posta; il nostro server SMTP controlla tutti i destinatari della mail e spedisce la mail a tutti i server SMTP degli utenti destinatari. A questo punto, ciascun server SMTP conserva la mail nella mailbox degli utenti destinatari, che saranno liberi di accedere alla mailbox in qualsiasi momento connettendosi al server POP3 o IMAP del proprio provider.



Dunque il processo di scambio di messaggi avviene tramite questi protocolli. Tuttavia spesso siamo abituati ad accedere alle nostre mailbox utilizzando un browser, che è un client HTTP. Come è possibile? Questo accade grazie al fatto che qualcuno (gli sviluppatori dei vari portali) ha creato per noi un'applicazione web (vale a dire: un sito internet) con la quale l'utente finale (noi) si interfaccia, mentre è il server web che comunica con i server mail (SMTP e POP3/IMAP). Questo viene fatto solo per consentire agli utenti finali un'interfaccia "comoda" come un'interfaccia web, ma lo scambio di messaggi avviene sempre e solo tramite i già citati protocolli, che ora analizzeremo in maggior dettaglio.

SMTP

Questo protocollo, definito nell'RFC 5321, costituisce il cuore vero e proprio del servizio di rete "Posta elettronica", e consiste di due parti. La prima parte che si occupa della ricezione della mail da parte dei client classici (es. gli user agent o server SMTP); la seconda parte che

invece si occupa di inoltrare i messaggi fra server SMTP di provider diversi. Risalente nella sua versione originaria al 1982 (molto prima dell'invenzione di HTTP), risente tuttavia di alcuni concetti "arcaici" che sono assolutamente inadatti alle caratteristiche di rete a cui siamo abituati ai giorni nostri, quasi 40 anni dopo la sua invenzione. La limitazione più importante è che i messaggi di posta elettronica (header e body) sono trattati come messaggi di testo ASCII a 7 bit. Se questa assunzione poteva avere senso negli anni '80 del XX secolo, oggi, in piena era multimediale, essa non ha più senso, in quanto richiede di ricorrere alla codifica di un qualsiasi contenuto diverso da testo semplice in ASCII e di metodi per dire agli User Agent (i client di posta elettronica, come l'applicazione di Gmail, o Thunderbird, o Mail) come interpretare i byte così ottenuti.

Illustriamo un tipico scenario di utilizzo di SMTP. Supponiamo che Alice voglia mandare a Bob un semplice messaggio ASCII.

1. Alice (il cui indirizzo ad esempio è alice@outlook.it), tramite il proprio user agent, fornisce l'indirizzo di Bob (es. bob@gmail.com), compone il messaggio e clicca sul pulsante "Invia messaggio" del suo User Agent
2. Lo user agent di Alice invia il messaggio al suo mail server (quindi il server SMTP smtp.office365.com, il provider di Alice); al suo interno il messaggio viene collocato in una coda di messaggi
3. Il mail server di Alice, arrivato nella coda il turno di inoltrare il messaggio di Alice, apre una connessione TCP con il server SMTP del provider di Bob (che è un programma che è in ascolto di connessioni e si trova sull'host smtp.gmail.com)
4. Dopo un handshaking SMTP (vale a dire un semplice scambio di messaggi di presentazione/autenticazione), il client SMTP invia il messaggio di Alice sulla connessione TCP.
5. Presso il mail server di Bob, il lato server di SMTP riceve il messaggio, che viene posizionato nella casella di Bob.
6. Bob, quando lo ritiene opportuno, invoca il proprio user agent per leggere il messaggio.

È importante osservare che di solito SMTP non usa mail server intermedi per inviare la posta, anche quando questi sono collocati agli angoli opposti del mondo. Se il server di Alice si trova a Hong Kong e quello di Bob a New York, la connessione TCP ha luogo direttamente tra i server presenti nelle due città. In particolare, se il mail server di Bob è spento, il messaggio rimane nel mail server di Alice e attende un nuovo tentativo. Il messaggio non viene posizionato in alcun mail server intermedio.

Osserviamo attentamente il trasferimento SMTP di un messaggio da un mail server di posta a un altro. Scopriremo che il protocollo SMTP presenta molte somiglianze con i protocolli usati per l'interazione umana faccia a faccia. Primo, il client SMTP (in esecuzione sul mail server di invio) fa stabilire a TCP una connessione sulla **porta 25** verso il server SMTP (in esecuzione sul mail server in ricezione). Se il server è inattivo, il client riprova più tardi. Una volta stabilita la connessione, il server e il client effettuano una qualche forma di "riconoscimento" a livello applicativo. Proprio come le persone che si presentano prima di scambiarsi informazioni, client e server SMTP si presentano prima di effettuare scambi di informazioni. Durante questa fase, il client indica l'indirizzo e-mail del mittente (la persona che ha generato il messaggio) e quello del

destinatario. Dopo la reciproca presentazione, il client invia il messaggio. SMTP può contare sul servizio di trasferimento dati affidabile proprio di TCP per recapitare il messaggio senza errori. Nel caso avesse altri messaggi da inviare al server, il client ripeterebbe questo processo sulla stessa connessione TCP; viceversa, ordinerebbe a TCP di chiudere la connessione.

Diamo ora uno sguardo a un esempio di trascrizione di messaggi scambiati tra un client SMTP (C) e un server SMTP (S). Il nome dell'host del client è **`alice`** mentre il nome dell'host del server è **`gmail.com`**. Le righe di testo ASCII precedute da C: sono esattamente i comandi inviati dal client, mentre le righe precedute da S: sono esattamente le risposte inviate dal server. La seguente transazione inizia appena si stabilisce la connessione TCP:

```
S: 220 gmail.com
C: HELO alice
S: 250 Hello alice, pleased to meet you
C: MAIL FROM: <alice@gmail.com>
S: 250 alice@gmail.com  Sender ok
C: RCPT TO: <bob@libero.it>
S: 250 bob@libero.it    Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Ti piace il ketchup?
C: Che cosa ne pensi dei cetrioli?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 gmail.com closing connection
```

Nell'esempio sopra riportato, il client invia un messaggio ("Ti piace il ketchup? Che cosa ne pensi dei cetrioli?") dal server di posta gmail.com al server di posta libero.it. Come parte del dialogo, il client ha inviato cinque comandi: **HELO** (abbreviazione di HELLO), **MAIL FROM**, **RCPT TO**, **DATA** e **QUIT**. Questi comandi sono abbastanza auto-esplicativi traducendoli dall'inglese ("ciao", "mail da", "recapitare a", "dati", "esci"). Il client invia anche una riga che consiste unicamente di un punto, che indica al server la fine del messaggio. Il server invia risposte a ogni comando, e ciascuna presenta un codice di risposta e qualche spiegazione (opzionale) in inglese. Ricordiamo che SMTP fa uso di connessioni persistenti: se il mail server di invio ha molti messaggi da inviare allo stesso mail server in ricezione, può mandarli tutti sulla stessa connessione TCP. Per ciascun messaggio il client inizia il processo con un nuovo MAIL FROM: crepes fr e stabilisce la fine del messaggio con un punto isolato. Il comando QUIT viene inviato solo dopo aver spedito tutti i messaggi.

Formato di un messaggio di posta elettronica

Quando Alice scrive una lettera di posta ordinaria a Bob, potrebbe includere come intestazione tutte le informazioni accessorie in cima alla lettera, tra cui l'indirizzo di Bob, 1' indirizzo del mittente e la data. Analogamente, il corpo dei messaggi di posta elettronica è

preceduto da un'intestazione contenente informazioni di servizio. Tale informazione periferica è contenuta in una serie di righe di intestazione, definite nell'RFC 5322. Queste righe sono separate dal corpo del messaggio mediante una riga senza contenuto. L'RFC 5322 specifica il formato esatto per le righe di intestazione (**header**) della posta e la loro interpretazione. Come avviene per HTTP, queste righe contengono testo leggibile, costituito da una parola chiave seguita da due punti a loro volta seguiti da un valore. Alcune parole chiave sono obbligatorie, mentre altre sono opzionali. Ogni intestazione, ad esempio, deve avere una riga **`From:`** e una riga **`To :`**. Un'intestazione può includere una riga **`Subject :`** e altre righe di intestazione opzionali. È importante osservare che queste righe sono differenti dai comandi SMTP analizzati nel paragrafo precedente (anche se contengono alcune parole comuni quali "from" e "to"). I comandi di tale paragrafo facevano parte del protocollo SMTP; le righe di intestazione esaminate qui sono invece parte del messaggio stesso. È molto importante capire che **un messaggio e-mail è formato interamente da caratteri ASCII, anche quando all'interno del messaggio sono allegati dei file binari (immagini, video, pdf)**.
Ecco una tipica intestazione di messaggio.

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Alla ricerca del significato della vita.
```

Dopo l'intestazione, segue una riga vuota; quindi troviamo il corpo del messaggio (in ASCII).

```
Essere, o non essere, questo è il dilemma:
se sia più nobile nella mente soffrire
colpi di fionda e dardi d'atroce fortuna
o prender armi contro un mare d'affanni
e, opponendosi, por loro fine?
```

Di seguito indichiamo alcuni header notevoli oltre a quelli di base già visti.

Content-Type

Questo header è molto importante. Indica infatti il tipo MIME (Multipurpose Internet Mail Extension) del corpo del messaggio di posta elettronica. Un MIME type è semplicemente una stringa che indica come bisogna interpretare un certo flusso di byte. Il formato di un MIME type è molto semplice: `oggetto/formato`

dove al posto di `oggetto` vi è una parola chiave che specifica il tipo di oggetto (es. `text`, `image`...) e al posto di `formato` vi è una parola chiave che specifica il formato (ad esempio, se l'oggetto è un testo, `plain`, html...). Ogni coppia oggetto/formato costituisce un tipo MIME (MIME type o content-type); alcuni MIME types validi sono:

- text/plain: indica un formato di testo ASCII semplice, senza allegati
- text/html: indica un messaggio formato da tag HTML
- application/pdf: indica un flusso di byte corrispondente ad un file pdf

- image/jpeg: indica un flusso di byte corrispondente ad un’immagine jpg
- image/png: indica un flusso di byte corrispondente ad un’immagine png
- video/mp4: indica un flusso di byte corrispondente ad un video mp4

La necessità di avere un MIME type specificato per i messaggi di posta elettronica si deve proprio al fatto che un messaggio è formato da caratteri ASCII. Quindi se vogliamo rendere più “ricchi” i nostri messaggi di posta (ad esempio arricchendoli con immagini, video, o più in generale allegati), dobbiamo specificare che all’interno di un messaggio, un determinato flusso di byte deve essere interpretato nel modo corretto.

Return-Path

È un header che gli user agent utilizzano per implementare la funzione “Rispondi a”. In esso viene specificato l’indirizzo e-mail a cui rispondere, che potrebbe anche non coincidere con il mittente.

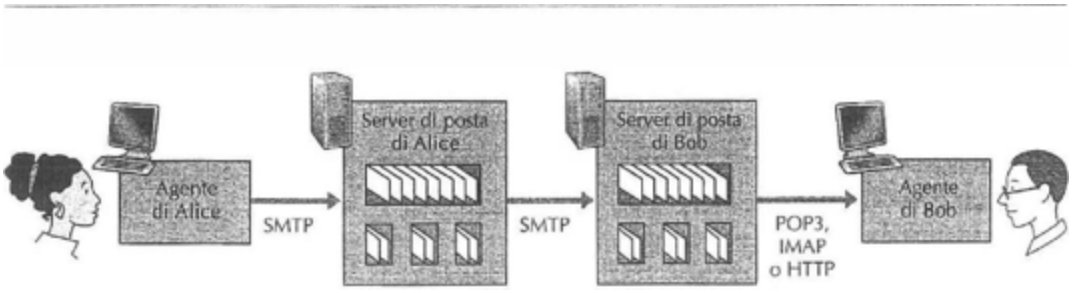
Accesso alla mailbox

Quando SMTP consegna il messaggio di Alice al mail server destinatario, questo lo colloca nella casella di posta di Bob. Per tutta la trattazione abbiamo tacitamente ipotizzato che Bob legga la propria posta collegandosi all'host che svolge la funzione di server ed eseguendo un programma di lettura. Fino ai primi anni '90 questo era il metodo standard. Attualmente, anche l'*accesso* alla posta elettronica utilizza un'architettura client-server: l'utente legge le e-mail con un client in esecuzione sul proprio sistema periferico (il PC dell'ufficio, un laptop o uno smartphone). Eseguendo un client di posta su un PC locale, gli utenti beneficiano di molteplici possibilità, tra cui la capacità di visualizzare messaggi multimediali e allegati (questo grazie all’header Content-Type presente fra le intestazioni dei messaggi di posta).

Un tipico utente manda in esecuzione uno user agent sul PC locale, ma accede alla propria casella memorizzata su un mail server condiviso con altri utenti e sempre attivo. Questo server è generalmente gestito dal provider (fornitore) di servizi di posta dell'utente (per esempio, gmail.com, o yahoo.it).

Consideriamo ora il percorso che un messaggio di posta intraprende quando viene spedito da Alice a Bob. Abbiamo appena imparato che, in qualche punto del percorso, il messaggio deve essere depositato nel mail server del provider Bob. Questo si potrebbe fare semplicemente forzando lo user agent di Alice a spedire messaggi direttamente al mail server di Bob e tutto questo potrebbe essere conseguito da SMTP. Tuttavia, lo user agent del mittente non dialoga in modo diretto con il server del destinatario. Piuttosto, lo user agent di Alice utilizza SMTP per recapitare i messaggi (PUSH) in ordine di tempo al suo server SMTP, al server SMTP di Bob. Qual è il motivo di questa procedura in due fasi? Principalmente perché, se non utilizzasse il suo mail server come punto intermedio, lo user agent di Alice non saprebbe come gestire un mail server di destinazione non raggiungibile. Alice deve dapprima depositare la e-mail nel proprio mail server, che può ripetutamente tentare l'invio del messaggio al mail server di Bob, per esempio ogni trenta minuti, finché questo non diventa operativo. E nel caso in cui il proprio server di posta sia inattivo, Alice può lamentarsi con il proprio amministratore di sistema! L'RFC relativa a SMTP definisce i comandi SMTP per consegnare un messaggio attraverso più server SMTP.

Tuttavia nel nostro puzzle manca ancora un pezzo. Come fa un destinatario (quale Bob), che esegue uno user agent sul proprio PC locale, a ottenere i messaggi che si trovano nel mail server del suo provider? Osserviamo che lo user agent di Bob non può usare SMTP per ottenere tali messaggi dato che si tratta di un'operazione di PULL, mentre SMTP è un protocollo di push. Il puzzle viene completato introducendo uno speciale protocollo di accesso alla posta, che trasferisce i messaggi dal mail server di Bob al suo PC locale. Attualmente esistono svariati protocolli del genere, tra cui post office protocol — versione 3 (POP3), Internet mail access protocol (IMAP) e HTTP.



La figura fornisce un riassunto dei protocolli di posta su Internet: SMTP è usato per trasferire posta dal server del mittente a quello del destinatario ed è anche utilizzato per trasferire la posta dallo user agent al mail server del mittente. Per trasferire messaggi dal mail server allo user agent del destinatario viene impiegato un protocollo di accesso alla posta, quale POP3.

POP3

POP3 è un protocollo di accesso alla posta estremamente semplice, definito nell'RFC 1939, che è breve e di agevole lettura. Dato che il protocollo è tanto semplice, le sue funzionalità sono piuttosto limitate. POP3 entra in azione quando lo user agent (il client) apre una connessione TCP verso il mail server (il server) sulla porta 110. Quando la connessione TCP è stabilita, POP3 procede in tre fasi: **autorizzazione**, **transazione** e **aggiornamento**. Durante la prima fase (autorizzazione) lo user agent invia nome utente e password (in chiaro) per autenticare l'utente. Durante la seconda fase (transazione) lo user agent recupera (elenca e legge) i messaggi; inoltre, durante questa fase, può *marcare* i messaggi per la cancellazione, rimuovere i marcatori di cancellazione e ottenere statistiche sulla posta. La fase di aggiornamento ha luogo dopo che il client ha inviato il comando QUIT, che conclude la sessione POP3; in questo istante, il server di posta rimuove i messaggi che sono stati marcati per la cancellazione.

In una transazione POP3 lo user agent invia comandi e il server reagisce a ogni comando con una tra due possibili risposte: +OK (talvolta seguito da dati dal server al client), usato dal server per indicare che il precedente comando andava bene; e -ERR, utilizzato dal server per indicare che qualcosa non ha funzionato nel precedente comando. POP3 è quindi un protocollo in cui **non vengono utilizzati gli status code nelle risposte**.

La fase di autorizzazione ha due principali comandi: **`user <username>`** e **`pass <password>`**. Per comprenderli meglio, vi suggeriamo di adoperare telnet direttamente con un server POP3 usando la porta 110 e inviare i comandi. Supponiamo che il vostro mail server si chiami `mailServer.it`. Vedremo qualcosa che assomiglia a:

```
telnet mailServer.it 110
+OK POP3 server ready user bob
+OK
pass hungry
+OK user successfully logged on
```

Nell'interazione precedente in grassetto troviamo le risposte del server e in carattere normale le richieste del client. Se sbagliamo a scrivere un comando, il server POP3 risponderà con un messaggio - ERR. Questo scambio di messaggi fa parte della fase di **autenticazione** ad un server POP3.

Prendiamo ora in considerazione la fase di **transazione**. Uno user agent che usa POP3 può spesso essere configurato (dall'utente) per "scaricare e cancellare" o per "scaricare e mantenere". La sequenza di comandi inviati da uno user agent POP3 dipende dal modo operativo scelto. Nella modalità "scarica e cancella" lo user agent manderà i comandi `list`, `retr` e `dele`. Per esempio, supponiamo che l'utente abbia due messaggi nella propria casella di posta. Nel dialogo che segue, C : (client) è lo user agent e S : è il mail server:

```
C: list
S: 1 498
S: 2 912
S:
C: retr 1
S: ( bla bla……
…… bla)
S:
C: dele 1
C: quit
```

Lo user agent chiede innanzitutto al server di elencare la dimensione dei messaggi memorizzati, quindi recupera e cancella ogni messaggio dal server. Si noti che dopo la fase di autorizzazione, lo user agent ha utilizzato solo quattro comandi: list, retr, dele e quit. La sintassi è definita nell' RFC 1939. Una votla richiesta la lista dei messaggi (tramite il comando list), il server risponde ocn l'elenco dei messaggi, costituito da una lista formata da una coppia di numeri: il primo costituisce un identificativo numerico che verrà utilizzato per leggere il messaggio con il comando RETR, mentre il secondo costituisce la dimensione in byte del messaggio. Una volta elaborato il comando quit, il server POP3 entra nella fase di **aggiornamento** e rimuove il messaggio 1 dalla casella di posta. Un problema legato alla modalità "scarica e cancella" è che il destinatario, Bob, potrebbe voler accedere ai propri messaggi di posta da più macchine, per esempio dal PC del suo ufficio, dal computer di casa o dal portatile. La modalità "scarica e cancella" ripartisce i messaggi di posta di Bob sulle tre macchine; in particolare, se Bob legge dapprima un messaggio sul PC del suo ufficio, non sarà in grado di rileg-gerlo dal portatile. In modalità "scarica e mantieni", lo user agent lascia i

messaggi sul server di posta dopo averli scaricati. In questo caso, Bob può rileggere i messaggi da diverse macchine, in differenti momenti. Durante una sessione tra uno user agent e il mail server, il server POP3 mantiene alcune informazione di stato; in particolare, tiene traccia dei messaggi dell'utente marcati come cancellati. Tuttavia, il server POP3 non trasporta informazione di stato tra sessioni POP3. Questa mancanza di informazione di stato persistente tra sessioni semplifica di molto l'implementazione.

Configurazione di un client di posta

Come ultima parte della trattazione del servizio di rete “Posta elettronica”, vediamo come è possibile configurare uno user agent di posta in maniera che possiamo utilizzarlo per mandare e spedire e-mail. Innanzitutto uno user agent è semplicemente un software che rispetta i protocolli SMTP e POP3. Questo significa che l’autore del software ha dovuto studiare le RFC menzionate in precedenza, il formato dei comandi che vengono scambiati fra client e server, il formato dei messaggi di posta elettronica, gli header, etc etc.

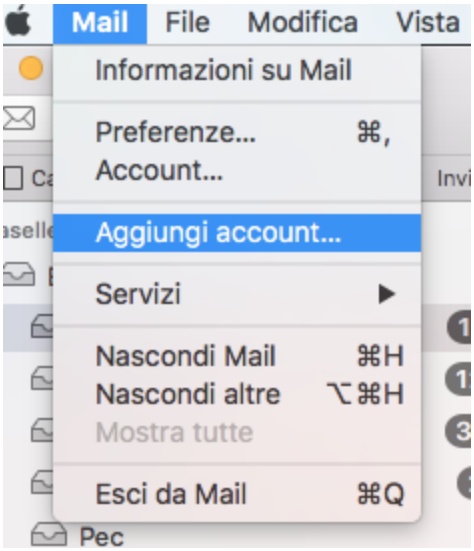
La fase di configurazione dello user agent è solitamente delegata all’utente, sebbene alcuni software abbiano un processo guidato per i provider di posta più famosi (gmail, outlook, yahoo, etc).

Innanzitutto, essendo il servizio di posta elettronica basato su due fasi ed essendo basato su due protocolli (SMTP e POP3/IMAP), è necessario recuperare le informazioni di connessione ai server di invio e ricezione del nostro provider. Ad esempio, se abbiamo gmail, una rapida ricerca sul web con una chiave come “Parametri di connessione posta gmail” dovrebbe essere sufficiente. Fra i primi risultati dovremmo trovare (oltre all’immancabile guida “for dummies” di Aranzulla) questo URL: https://support.google.com/mail/answer/7126229?hl=it .

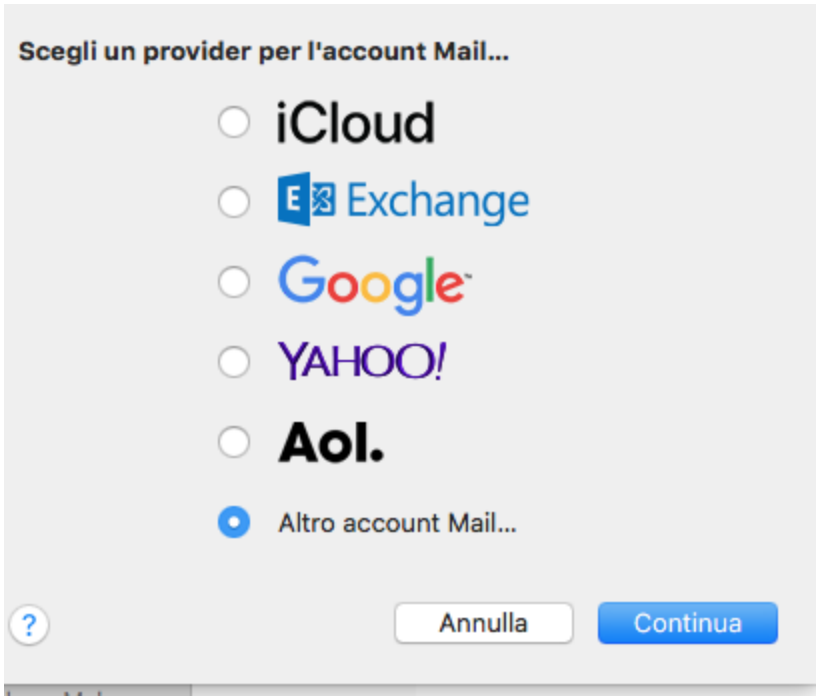
Leggendo un po’la pagina, troveremmo la tabella in figura successiva. Qui dentro c’è scritto tutto ciò di cui abbiamo bisogno per configurare un qualsiasi user agent di posta con il provider di email “gmail”.

Server posta in arrivo (IMAP)	imap.gmail.com Richiede SSL: Sì Porta: 993
Server posta in uscita (SMTP)	smtp.gmail.com Richiede SSL: Sì Richiede TLS: Sì (se disponibile) Richiede autenticazione: Sì Porta per SSL: 465 Porta per TLS/STARTTLS: 587
Nome completo o Nome visualizzato	Il tuo nome
Nome account, Nome utente o Indirizzo email	Il tuo indirizzo email completo
Password	La tua password Gmail

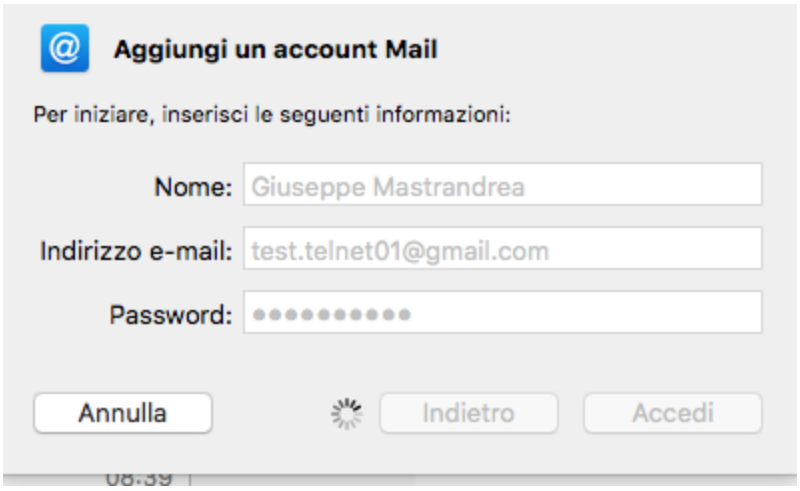
Ad esempio, supponiamo di aggiungere il nostro account gmail allo user agent “Mail” disponibile per Mac OS. Clicchiamo su “Aggiungi account”:



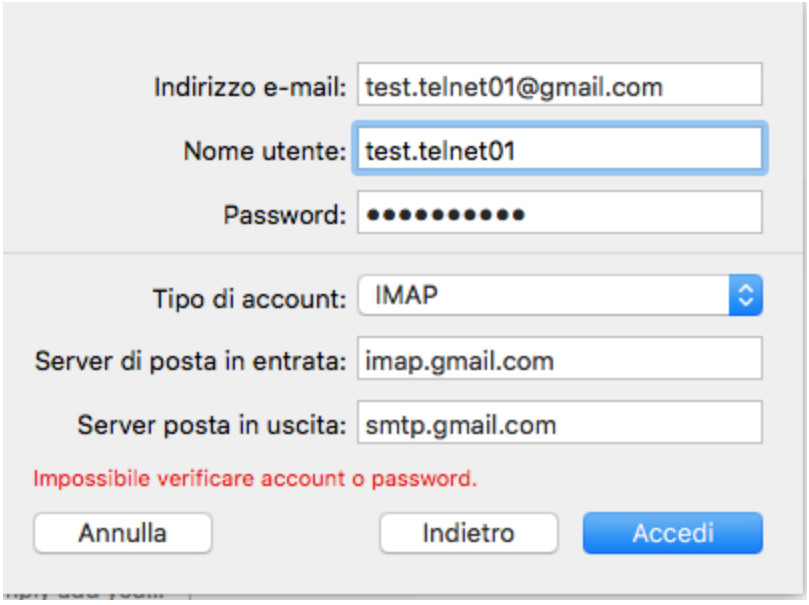
Nella finestra successiva, lo user agent ci chiederà quale account di posta vogliamo aggiungere. Google è già presente, ma siccome vogliamo generalizzare la procedura, scegliamo “Altro account email”:



Inseriamo successivamente le credenziali della mailbox e clicchiamo su “Accedi”:



Dopo, finalmente, potremo inserire i parametri di connessione che abbiamo cercato in precedenza:

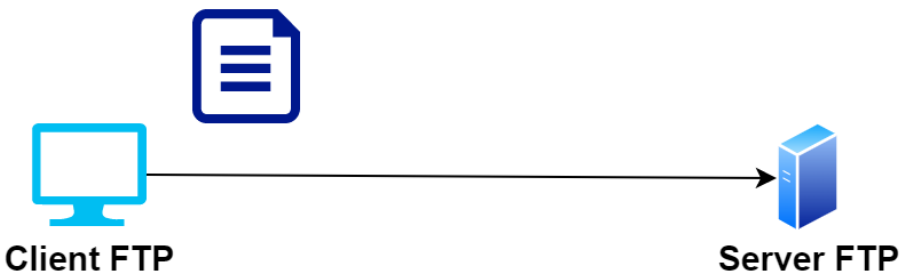


Come nome utente mettiamo il nome della mailbox priva del suffisso “@gmail.com”. Alcuni user agent ci chiederanno anche i numeri di porta e le informazioni sul certificato di sicurezza. Sono tutte informazioni che sono presenti nella tabella che abbiamo cercato poco su. Una volta inserite queste informazioni, basterà cliccare su “Accedi” per avere lo user agent correttamente configurato. Sarà quindi possibile leggere e inviare mail dal proprio user agent utilizzando il proprio account di posta.

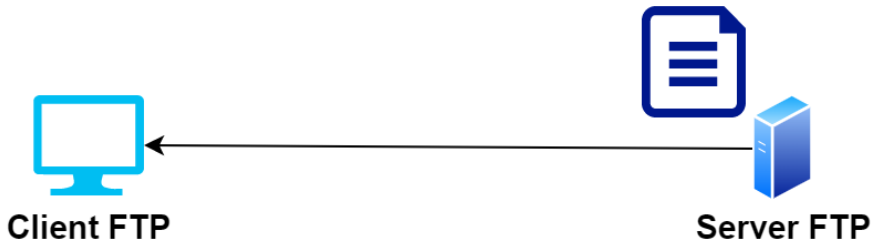
FTP

In una tipica sessione FTP, un utente utilizza un programma (client FTP) per trasferire file dal suo computer ad un computer remoto. **FTP** (File Transfer Protocol, RFC 959) è quindi un protocollo che implementa l’applicazione di rete **trasferimento file**. In sostanza abbiamo un client e un server che mettono in condivisione fra loro una porzione del loro file system. Il client potrà fare:

- **Upload** di un file quando dovrà passare un file al server



- **Download** di un file quando dovrà scaricare un file dal server



FTP è un protocollo affidabile, quindi a livello di trasporto si basa su TCP. Questo significa che prima dell'inizio della sessione FTP c'è bisogno di instaurare la connessione. Quando il processo client si connette ad un server FTP, quindi, per prima cosa viene creato un socket TCP. Peculiarità del protocollo FTP è di utilizzare 2 connessioni per fornire il servizio di trasferimento file, quindi due socket.

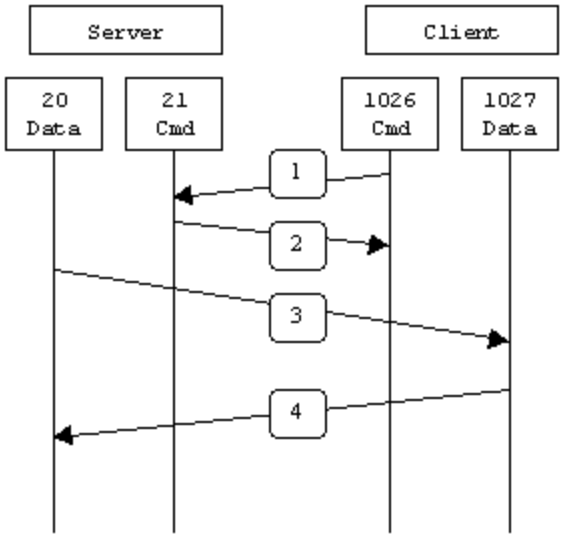
La prima connessione è detta **connessione di controllo**; in essa il demone del server è in ascolto sulla porta 20, ed è utilizzata per lo scambio dei comandi FTP (ne vedremo qualcuno più avanti). La seconda connessione è detta **connessione dati**, ed è usata per il trasferimento file vero e proprio. In essa il demone del server è in ascolto sulla porta 21.

Una tipica sessione FTP si svolge in questo modo:

- Tramite comando OPEN ci si connette ad un server FTP
- Si passano al server FTP username e password del proprio utente tramite i comandi USER e PASS
- Si richiede l'elenco della directory corrente
- Si richiede il download di un file tramite il comando RETR
- Si scarica il file

Ad ogni punto dei precedenti corrisponde uno scambio di messaggi fra client e server. Di queste interazioni fra client e server solo l'ultima avviene sulla connessione dati, mentre tutte le altre avvengono sulla connessione di controllo.

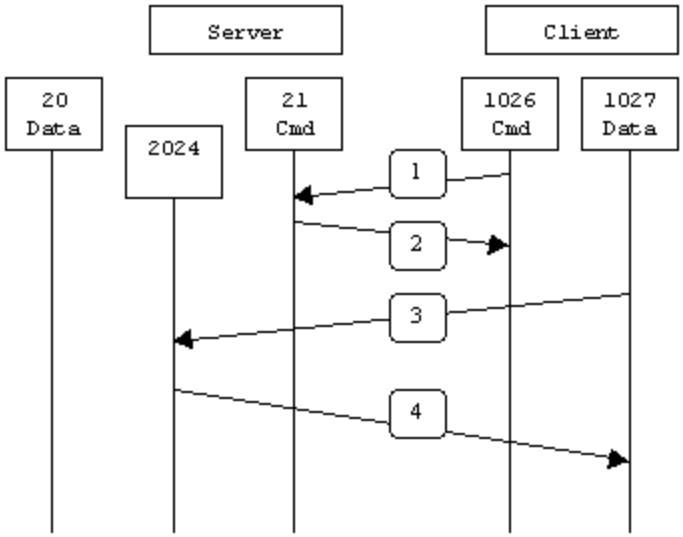
In FTP abbiamo due modalità di trasferimento dei file: modalità attiva e passiva[4]. Entrambe le modalità si riferiscono al solo trasferimento dati. Nella modalità **attiva**, supponendo che il client usi una porta N (con N > 1023) per la connessione di controllo, arrivato il momento di trasferire un file, esso "apre" la porta N + 1, la invia al server tramite il comando PORT N+1, e si mette in attesa. Il server, ricevuto il numero di porta come payload del comando PORT, inizializza il socket connettendosi all'IP e alla porta del client, mentre come porta del server avremo la porta 21. Ecco quel che accade nella modalità attiva:



Allo step 1, il client (che per ipotesi qui utilizza il numero 1026 come porta di controllo) contatta sulla connessione di controllo il server (quindi sulla sua porta 20), inviando il comando PORT 1027. Il server risponde con uno status code 2YZ nello step 2. Nello step 3, il server crea una connessione sulla sua porta 21, andando a “bussare” alla porta 1027 del client, che allo step 4 risponderà con un messaggio di conferma.

Il problema principale della modalità attiva è sul client. Per ragioni di sicurezza, infatti, è molto probabile che il firewall del client (un software capace di rigettare richieste di connessioni provenienti dall’esterno per aumentare la sicurezza del sistema su cui è installato) blocchi le connessioni in entrata, quindi anche quella del server FTP, rendendo impossibile il trasferimento dei file.

Per aggirare questo problema, è ormai prassi consolidata utilizzare la modalità **passiva** di FTP. Semplicemente, in questa modalità, è il client ad iniziare la connessione dati, oltre a quella di controllo. Quando viene aperta una connessione FTP il client apre due porte adiacenti > 1023. La prima porta contatta il server sulla porta 21, ma invece di utilizzare un comando PORT e attendere di essere contattato, il client utilizzerà sulla connessione di controllo il comando PASV (senza argomenti). Il server risponderà con un numero P di porta, che sarà usato dal client per inizializzare la connessione dati e trasferire i file. Nella figura successiva un esempio di modalità FTP passiva.



Nello step 1 il client contatta il server sulla porta di controllo mandandogli il comando PASV. Il server risponde nello step 2 con numero di porta (es. 2024), dicendo al client che si mette in ascolto su quella porta. Nello step 3 il client quindi inizializza la connessione dati connettendosi alla porta 2024 del server. Nello step 4 il server risponde con un ACK al client, e il trasferimento può cominciare.

Ricapitolando:

- FTP modalità **attiva**:
  - controllo : client >1023 → server 20
  - dati : client >1023 ← server 21

- FTP modalità **passiva**:
  - controllo : client >1023 → server 20
  - dati : client >1024 → server >1023

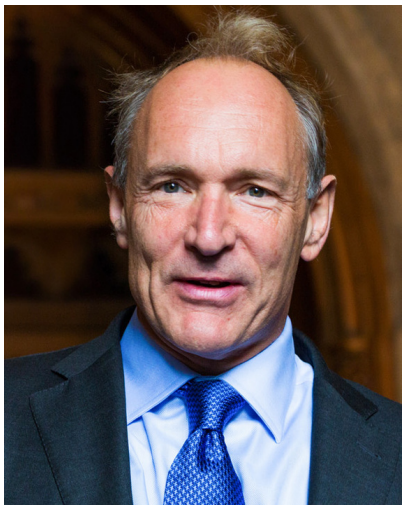
HTTP

Nell'immaginario comune, il Web e "Internet" sono la stessa cosa. Pensate un attimo a dialoghi che quotidianamente ascoltiamo o di cui siamo (ahinoi) partecipi: "Da dove hai comprato quel maglione?" - "L'ho preso da Internet.", oppure "Ricordi qual è la capitale del Portogallo?" - "No, guardo su Internet", o ancora "Che buona questa ricetta!" "Vero? L'ho trovata su Internet!", e molte altre cose ancora.

Arrivati a questo punto della nostra formazione, dovremmo capire che dire semplicemente "L'ho trovato su Internet" è un'espressione quantomeno impropria, che farebbe rabbrividire persino uno studente del primo anno di ingegneria delle telecomunicazioni. Dovremmo già sapere che *Internet è l'infrastruttura a livello di rete* che permette di interconnettere fra loro computer (host) con un certo indirizzo di rete (indirizzo IP). Al contrario, il **World Wide Web**

(abbreviata con l'acronimo **www**), o semplicemente *Web* è invece *l'applicazione di rete che ci consente di consultare, scambiare, creare risorse in rete*. Un po'come la posta elettronica, il Web è semplicemente un altro modo con cui utilizziamo la rete Internet.

Il Web nasce nei primi anni '90 dello scorso secolo, da una brillante idea di **Tim Berners-Lee**, un (allora) ricercatore universitario inglese che nel 1989 aveva pubblicato il paper scientifico "Information Management: A Proposal"[5]. In esso Berners-Lee propone un metodo per gestire al meglio le informazioni scientifiche all'interno del CERN. Nonostante fosse solo una proposta del tutto teorica, all'interno del documento si fa riferimento ad alcuni requisiti che poi sarebbero diventati le caratteristiche principali del Web come noi lo conosciamo: accesso remoto ai dati, trasversalità dei sistemi di accesso alle informazioni, presenza di link. Soprattutto, si fa riferimento ad una tecnologia nuova per l'epoca: l'utilizzo degli **ipertesti**, documenti di testo semplice "aumentati" con la presenza di collegamenti ad altri documenti o ad altre risorse.



Tim Berners-Lee, il padre del Web e di HTTP

Nel 1991, lo stesso Berners-Lee sviluppò la prima versione del protocollo che avrebbe dovuto implementare il Web come descritto in precedenza: **HTTP** (**Hyper Text Transfer Protocol**) nella sua versione **0.9**. Come dice il nome, HTTP è il *protocollo che detta le regole per trasferire ipertesti (e quindi informazioni) su una rete*. Berners-Lee, grazie alla sua invenzione, ha vinto svariati premi in ambito accademico e riconoscimenti anche civili, tanto che esiste una pagina Wikipedia con l'elenco delle onoreficenze ricevute. Tuttavia, al contrario di quello che si potrebbe pensare, non è mai diventato ricco, nonostante la sua invenzione abbia rivoluzionato la società e la vita di tutti gli abitanti della Terra. Perchè? Be', fondamentalmente per amore della scienza: "Se avessi voluto essere pagato per la mia invenzione, oggi non esisterebbe alcun World Wide Web, ma tanti piccoli Web, nessuno dei quali gratuito"[6], dice alla consegna del premio "Millenium Technology Prize", ad Helsinki nel 2004.

Prima di affrontare in dettaglio HTTP soffermiamoci brevemente sulla terminologia web.

Una pagina web (web page), detta anche *documento*, è costituita da *oggetti*. Un oggetto è semplicemente un file (quale un file HTML, un'immagine JPEG, un' applet Java, una clip video e così via) indirizzabile tramite un **URL**: **Uniform Resource Locator**. La maggioranza delle

pagine web consiste di un file HTML principale e diversi oggetti referenziati da esso. Per esempio, se una pagina web contiene testo in HTML e cinque immagini JPEG, allora la pagina nel complesso presenta sei oggetti: il file HTML più le cinque immagini. Il file HTML referenzia gli altri oggetti nella pagina tramite il loro URL. Vediamo un esempio di URL (l'immagine che contiene il logo di una nota testata giornalsitica):

https://static.internazionale.it/assets/img/internazionale-logo.png

Esaminiamo ogni parte di questo URL. Abbiamo, da sinistra a destra:

- Il protocollo di accesso alla risorsa, in questo caso https:// (una versione di http che implementa un livello di crittografia)
- Un nome di host (static.internazionale.it), che rappresenta l'host remoto (banalmente: il sito) su cui si trova la risorsa che vogliamo reperire
- Il percorso della risorsa all'interno dell'host (assets/img/)
- Finalmente, il nome della risorsa completa di estensione (internazionale-logo.png)

Aggiungiamo che fra il nome dell'host e il path della risorsa c'è, sottinteso, il numero di porta. Nel nostro caso il numero di porta sarebbe 80, visto che stiamo utilizzando HTTP per accedere alla risorsa "internazionale-logo.png". Quindi l'URL completo della risorsa in teoria sarebbe:

https://static.internazionale.it:80/assets/img/internazionale-logo.png

Un browser web (come Google Chrome o Firefox) implementa il lato client di HTTP (quando parliamo di Web useremo le parole *browser* e *client* in modo intercambiabile). Un web server, che implementa il lato server di HTTP, ospita oggetti web, indirizzabili tramite URL. Tra i più popolari ricordiamo Apache, nginx, node.js e Microsoft Internet Information Server.

Architettura

HTTP è un protocollo client-server la cui prima versione ufficiale (1.0) risale al 1996: l'RFC 1945. A prima vista HTTP è un semplice protocollo di testo basato su TCP/IP. In particolare, dovendo offrire un servizio affidabile, HTTP, utilizza TCP a livello di trasporto. Il server, in HTTP, non è altro che un software in ascolto sulla porta 80 che è in grado di capire il linguaggio definito dall'RFC. Lo standard HTTP/0.9 originale non prevedeva alcuno spazio per lo scambio di metadati aggiuntivi tra le parti coinvolte. La richiesta del client consisteva sempre di un'unica riga, che inziava con GET seguito dal percorso e dalla stringa di query dell'URL e terminata con un singolo avanzamento di riga CRLF (Carriage Return Line Feed, in sostanza un "a capo"):

```
GET /fuzzy_bunnies.html
```

In risposta a questo messaggio, il server rispondeva con il codice HTML della risorsa cercata. Non molto eccitante, vero? È stato infatti con l'avvento di HTTP 1.0 e (soprattutto) 1.1 che sono state introdotte tutte le novità che conosciamo e a cui siamo abituati. HTTP 1.0, ad

esempio, introdusse l'utilizzo di alcuni metadati da associare alla richiesta del file vero e proprio: alla prima riga della richiesta viene aggiunta la versione del protocollo utilizzato, di seguito potranno esserci zero o più coppie `chiave: valore` (note come **header**) ciascuna su una riga a sé. Vengono inoltre introdotte altre operazioni da fare sulle risorse (mentre in origine era possibile solo fare GET, cioè lettura, di una risorsa). Ad esempio, una tipica richiesta HTTP 1.1 potrebbe essere la seguente:

```
   POST /fuzzy_bunnies/bunny_dispenser.php HTTP/1.1
   Host: www.fuzzybunnies.com
   User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130
Safari/537.36
   Content-Type: text/plain
   Content-Length: 17
   Referer: http://www.fuzzybunnies.com/main.html
   I REQUEST A BUNNY
```

Nello snippet precedente, ad esempio, abbiamo richiesto una risorsa chiamata bunny_dispenser.php, presente sul sito www.fuzzybunnies.com nella cartella fuzzy_bunnies, ci connettiamo con un browser Google Chrome, e trasmettiamo una serie di dati (la stringa I REQUEST A BUNNY) lunga 17 byte di tipo text/plain. Da dove abbiamo desunto tutte queste informazioni? Semplicemente dall'analisi degli header della richiesta, (Content-Type, Content-Length, User-Agent, Host, etc). Tutte queste informazioni permettono al server di migliorare la gestione della nostra richiesta, e poter rispondere in maniera molto più aderente alle esigenze del client.

È importante specificare fin da subito che il server invia i file richiesti ai client senza memorizzare alcuna informazione di stato a proposito del client. Per cui, in caso di ulteriore richiesta dello stesso oggetto da parte dello stesso client, anche nel giro di pochi secondi, il server procederà nuovamente all'invio, non avendo mantenuto alcuna traccia di quello precedentemente effettuato. Dato che i server HTTP non mantengono informazioni sui client, HTTP è classificato come protocollo senza memoria di stato (**stateless protocol**). Un web server è sempre attivo, ha un indirizzo IP fisso e risponde potenzialmente alle richieste provenienti da milioni di diversi browser.

Risorse

In generale, si può pensare ad HTTP come ad un protocollo il cui obiettivo è permettere lo *scambio di risorse in rete*. A questo punto dovrebbe sorgere spontanea una domanda: *che cos'è una risorsa?* La risposta a questa domanda può essere non così scontata come sembra. Una risorsa può essere infatti davvero qualsiasi cosa: un file HTML, un'immagine, un oggetto JSON, una stringa XML, anche solo una variabile. Nello scenario standard di HTTP, tuttavia, possiamo dire che una risorsa può essere fondamentalmente di 3 tipi, a seconda di dove quella risorsa viene generata. Consideriamo per il momento il solo caso in cui navighiamo sul Web utilizzando il nostro browser preferito (Firefox, Chrome, Safari, etc). In questo caso quando

andremo a digitare sulla barra degli indirizzi l'URL del sito che vogliamo visitare e premeremo invio, partirà una **richiesta HTTP** verso il server Web che ospita il sito. Il server riceverà la richiesta, la elaborerà e ci restituirà una **risposta HTTP** appropriata (nel prossimo paragrafo studieremo i formati di richiesta e risposta), ovvero contenente una risorsa. Quali tipi di risorsa può restituirci un server?

- Risorsa **statica**: a questo tipo di risorse appartengono tutti i file e le risorse statiche, cioè immutabili nel tempo, su cui non è possibile fare alcuna operazione se non richiederle; pensiamo ad un file HTML (senza script javascript), un'immagine PNG, un video, un file CSS, etc. Insomma, tutto ciò che non muta nel tempo.
- Risorsa **dinamica**: una risorsa è dinamica se non esiste "a prescindere", ma viene generata "al momento" dal server web (una risorsa che viene generata al volo lato server);
- Risorsa **attiva**: vale a dire una risorsa che viene generata dal server, ma che consiste di una porzione di codice eseguita dal client. Pensate ad una pagina HTML con del codice Javascript: in questo caso la risorsa (il file HTML, sia esso statico o dinamico) contiene del codice che verrà eseguito lato client, rendendo il client non più un mero consumatore di risorse ma anche una parte attiva nel servizio richiesto.

Formato della richiesta[7]

Le specifiche presenti nelle RFC 1945 e 2616 contengono il formato sia della richiesta HTTP che della risposta. Ecco un tipico messaggio di richiesta HTTP:

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0 (Linux; Android 6.0);
Accept: text/html
```

Osservandolo accuratamente, notiamo innanzitutto che il messaggio è scritto in testo ASCII, in modo che l'utente sia in grado di leggerlo: questo non dovrebbe sorprenderci, visto che in tutti i protocolli di livello applicativo visti finora i messaggi sono scritti in plaintext, ovvero in codice ASCII. Inoltre, notiamo che consiste di svariate righe, ciascuna seguita da un carattere di ritorno a capo (carriage return) e un carattere di nuova linea (line feed). In generale, i messaggi di richiesta possono essere costituiti da un numero indefinito di righe, anche una sola. La prima riga (`GET /somedir/page.html HTTP 1.1`) è detta **riga di richiesta** (**request line**) e quelle successive righe di intestazione (header lines). La riga di richiesta presenta tre campi: il campo metodo, il campo PATH della risorsa e il campo versione di HTTP. Il campo metodo può assumere diversi valori, tra cui GET, POST, HEAD, PUT e DELETE. La maggioranza dei messaggi di richiesta HTTP usa il metodo GET, adottato quando il browser richiede una risorsa identificata dal campo PATH. Il campo "versione" è auto esplicativo: nell'esempio, il browser, che implementa la versione HTTP/1.1, sta richiedendo l'oggetto `/somedir/page.html`.

Spendiamo due parole sul campo metodo. Esso rappresenta **il tipo di operazione che vogliamo effettuare su una determinata risorsa**. I metodi più importanti di HTTP sono 4:

POST, GET, PUT e DELETE. Ciascuno di essi rappresenta una delle 4 operazioni **CRUD** che è possibile in generale fare su una risorsa.

- **C**reate
- **R**ead
- **U**pdate
- **D**elete

Leggendo le prime lettere delle quattro operazioni elencate prima si ottiene proprio l'acronimo *CRUD*. Dunque, i metodi HTTP di norma (significa che questo utilizzo non è previsto dallo standard, ma è universalmente accettato) mappano le 4 operazioni in questo modo:

- Una POST corrisponde ad un'operazione CREATE (creazione di una nuova risorsa)
- Una GET corrisponde ad un'operazione READ (lettura di una risorsa)
- Una PUT corrisponde ad un'operazione UPDATE (modifica di una risorsa esistente)
- Una DELETE corrisponde ad un'operazione DELETE (cancellazione di una risorsa esistente)

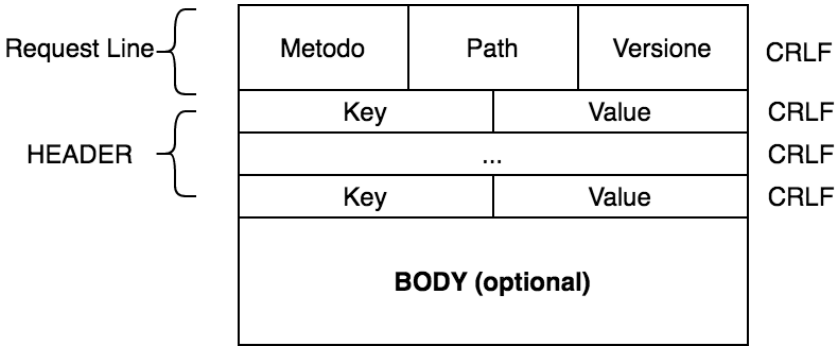
Pensiamo, ad esempio, a quando riempiamo un form online, per esempio un form di registrazione ad un sito. Di norma, quando clicchiamo sul pulsante di submit, *creiamo* un nuovo utente. Quindi il metodo utilizzato è il POST. Questo comportamento, tuttavia, non è obbilgatorio, e capita molto spesso che un form venga spedito in GET (pensiamo ad esempio ai form di ricerca, che quasi sempre utilizzano il metodo GET).

Ad ogni modo, torniamo ad esaminare la richiesta HTTP. Dopo la request line, cominciano gli **header della richiesta HTTP**. Essi hanno la stessa funzione degli header presenti nei messaggi di posta elettronica: arricchire con informazioni (spesso vitali) la richiesta HTTP. Come nei messaggi di posta, gli header HTTP sono delle coppie chiave/valore, e ciascun header occupa un rigo della richiesta HTTP. In questo caso, ad esempio, ci sono 4 header nella richiesta:

- **Host**: questo header specifica il nome di dominio del server verso cui ci stiamo connettendo; è possibile specificare all'interno di questo header anche il numero di porta su cui il server è in ascolto;
- **Connection**: connection controlla il comportamnto del socket TCP la ricezione della risposta alla richiesta attuale; in questo caso, il valore di questo header è `close`; questo significa che il client desidera che il socket venga chiuso dopo la risposta del server (sia essa positiva o negativa). Un altro possibile valore di Connection è `keep-alive`, che al contrario richiede al server di non avviare la chiusura TCP del socket dopo la risposta alla richiesta corrente, dando vita ad una *connessione persistente*; questo permetterebbe al client di effettuare altre richieste allo stesso sito sullo stesso socket senza dover effettuare una nuova fase di handshake.
- **User-Agent**: in questo header è contenuta una stringa che contiene informazioni sullo user agent del client: numero di versione, sistema operativo, etc etc. In questo caso, ad esempio, possiamo dire che lo User Agent è un browser mozilla Firefox installato su una macchina Android (per brevità questo header è stato tagliato).
- **Accept**: in questo header è contenuto il MIME-type della risorsa che il client si aspetta di ricevere in risposta alla richiesta.

Ci sono moltissimi altri header, alcuni solo di richiesta, altri solo di risposta, altri ancora validi sia per le richieste che per le risposte. Per un elenco dettagliato, consultate la pagina web https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.

Opzionalmente, solo nel caso di richieste in cui stiamo *trasmettendo* dei dati (es. nel caso di a richieste di creazione o modifica di risorse, ovvero con i metodi POST e PUT), è possibile trovare oltre alla request line e agli header, un **body della richiesta**, contenente i dati che si vogliono trasmettere al server. In generale, questo è il formato di una richiesta HTTP:



Con CRLF si indica Carriage Return Line Feed (praticamente fine riga/inizio riga). Le righe verticali non sono altro che spazi bianchi (il carattere ASCII U+0020).

Formato della risposta

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

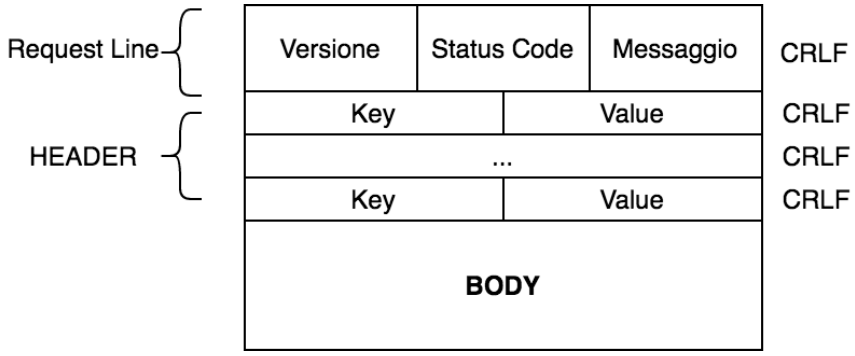
```
HTTP/1.1 200 OK
Connection: close
Date: Fri, 29 Feb 2020 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 09 Aug 2011 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html
<html>
…
</html>
```

Analizzando in dettaglio questo messaggio di risposta, osserviamo tre sezioni. Una riga di stato iniziale, sei righe di intestazione e il corpo. Quest'ultimo è il fulcro del messaggio: contiene l'oggetto richiesto (rappresentato in questo caso dal codice html della pagina che abbiamo richiesto). La **riga di stato** presenta tre campi: **la versione del protocollo**, uno **status code** e un corrispettivo **messaggio di stato**. In questo esempio, la riga di stato indica che il server sta usando HTTP/ 1.1 e che tutto va bene (ossia che il server ha trovato e sta inviando l' oggetto richiesto), come testimoniato dallo status code 200.

Osserviamo ora le righe di intestazione (gli **header della risposta**) ed analizziamo gli header di risposta:

- **Connection**: funziona come per la richiesta: il valore "`close`" è usato per comunicare al client che ha intenzione di chiudere la connessione TCP dopo l'invio del messaggio.
- **Date**: indica l'ora e la data di creazione e invio, da parte del server, della risposta HTTP. Si noti che non si tratta dell'istante in cui l'oggetto è stato creato o modificato per l'ultima volta, ma del momento in cui il server recupera l'oggetto dal proprio file system, lo inserisce nel messaggio di risposta e invia il messaggio.
- **Server**: indica che il messaggio è stato generato da un web server Apache; essa è analoga alla riga `User-Agent` nel messaggio di richiesta HTTP.
- **Last-Modified**: indica l'istante e la data il cui oggetto è stato creato o modificato per l'ultima volta. Tale riga è importante per la gestione dell'oggetto nelle cache, sia nel client locale sia in alcuni server in rete (proxy server o proxy).
- **Content-Length**: contiene la dimensione in byte dell'oggetto inviato.
- **Content-Type**: indica che l'oggetto nel corpo è testo HTML, e rappresenta il MIME Type della risorsa restituita al client (fa il paio con l'header di richiesta `Accept`).

Dopo l'esempio esaminiamo il formato generale di un messaggio di risposta:



Sull'ottimo sito MDN (Mozilla Developer Network), all'URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Status, è presente una lista degli status code HTTP più comuni, il consiglio è quello di dare un'occhiata almeno ai più famosi (200, 201, 206, 301, 302, 304, 400, 401, 403, 404, 405, 407, 500, 502, 504).

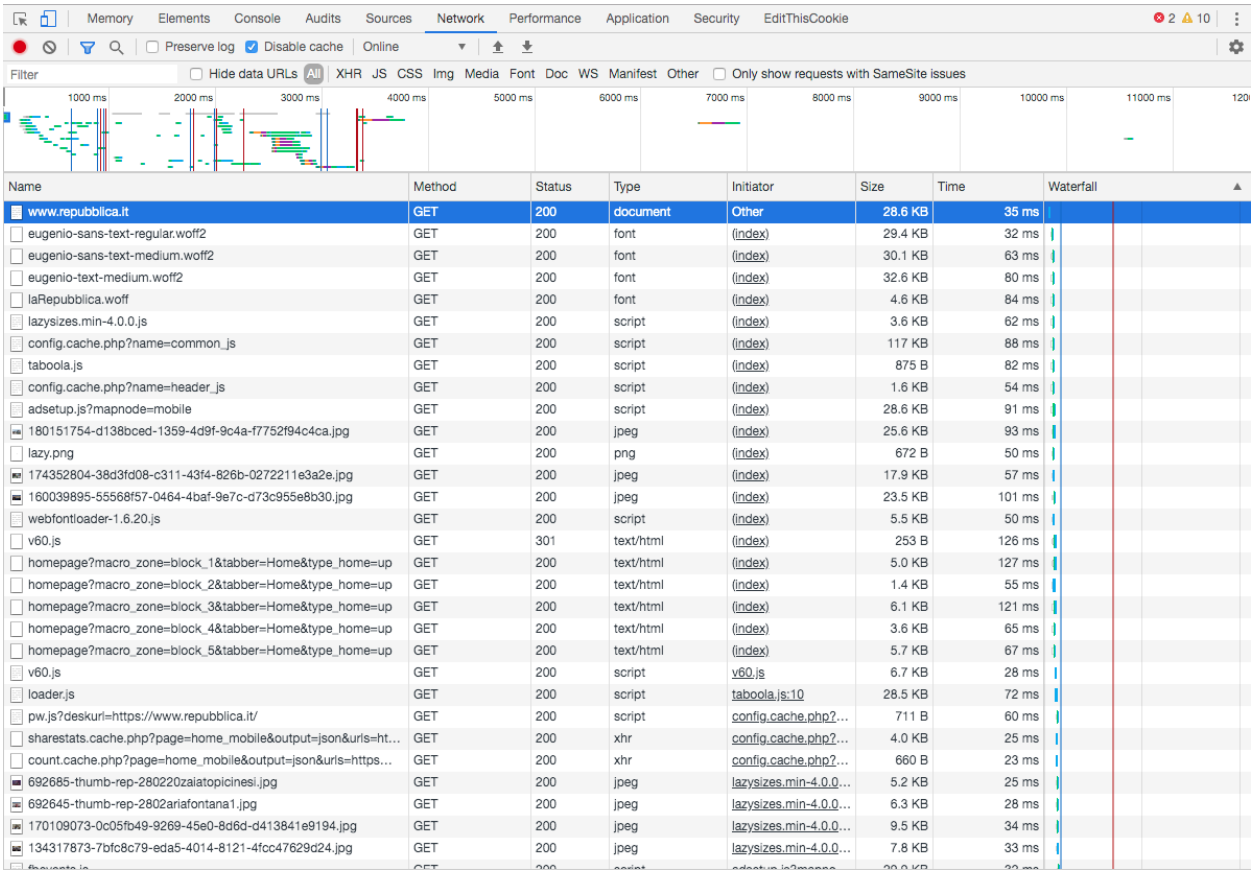
Un esempio di interazione HTTP

Navigare sul web è fin troppo facile e ormai iniziamo a darlo per scontato, ma a noi interessa vedere cosa *realmente* accade quando proviamo a connetterci ad un sito web. Per capire cosa succede abbiamo svariati modi, ne vedremo in seguito un paio.

Strumenti per sviluppatori del browser

Ogni browser moderno è dotato di strumenti per gli sviluppatori. Su Google Chrome, per esempio, è possibile attivare gli strumenti per sviluppatori con la combinazione di tasti CTRL+SHIFT+i. Si aprirà una finestra composta da varie schede. Quella che ci interessa è la

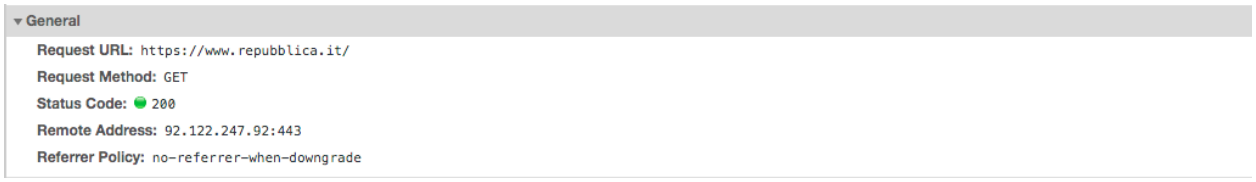
scheda “Network”: assicuriamoci di aprire prima gli strumenti per gli sviluppatori siano aperti prima che ci connettiamo ad un sito, o in caso siamo già connessi ricarichiamo semplicemente la pagina. Ad esempio, se ci connettiamo al sito https://www.repubblica.it, vedremo una serie di richieste HTTP come in screen:



Ogni riga in screen rappresenta una diversa richiesta fatta al server. Ogni riga in tabella è fatta in questo modo:

- Prima colonna: URL della richiesta
- Seconda colonna: Metodo della richiesta (in questo caso sono tutte GET)
- Terza colonna: Status code della risposta
- Quarta colonna: content-type della risposta (non per tutti è presente il MIME Type)
- Quinta colonna: chi ha generato la richiesta HTTP (ricordiamo che per ogni risorsa presente in un file HTML come immagini, file css, script, etc viene fatta una richiesta HTTP a parte)
- Sesta colonna: Dimensione in KB della risposta
- Settima colonna: tempo per ottenere la risposta

Andiamo a cliccare sulla prima riga per avere dettagli sulla richiesta/risposta HTTP corrispondente. Troveremo una sezione dedicata alle info generali:



una dedicata agli header di richiesta:

una dedicata agli header di risposta:

Comandi HTTP testuali

Un altro metodo per analizzare un flusso di comunicazione HTTP è quello di usare un client testuale, come telnet, o un comando come cURL. Nello screen di seguito un esempio di sessione telnet di collegamento allo stesso sito di prima:

```
MBP-di-Giuseppe:~ giuseppe$ telnet repubblica.it 80
Trying 213.92.16.101...
Connected to repubblica.it.
Escape character is '^]'.
GET / HTTP/1.1
Host: repubblica.it

HTTP/1.1 301 Moved Permanently
Date: Fri, 28 Feb 2020 17:53:27 GMT
Server: Apache
Location: http://www.repubblica.it/
Content-Length: 233
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://www.repubblica.it/">here</a>.</p>
</body></html>
```

In questo caso abbiamo semplicemente effettuato una richiesta di connessione al server web (porta 80) presente sull'host con nome "repubblica.it". Una volta connessi (messaggio "Connected to repubblica.it") abbiamo scritto a mano una semplice GET sul root path del sito

specificando 1.1 come versione di HTTP, e abbiamo aggiunto alla riga successiva (come dovrebbe essere adesso noto) l'header di richiesta obbligatorio Host. La risposta del server non si è fatta attendere, possiamo distinguere chiaramente status code, messaggio, versione di HTTP, header e infine il body della risposta: il codice HTML di una pagina che ci avvisa del redirect della risorsa.

Persistenza dello stato in HTTP

Abbiamo detto in precedenza che i server HTTP non mantengono informazioni sui client, e che quindi HTTP è classificato come protocollo senza memoria di stato (**stateless protocol**). Che significa esattamente questa affermazione? Di base, significa che ogni richiesta, anche quelle che vengono mandate sullo stesso socket (quindi dallo stesso client), viene trattata in maniera indipendente. Non c'è nessuna relazione fra 2 richieste mandate (ad esempio) in rapida successione, anche se fatte sullo stesso socket. Ora, la domanda *dovrebbe* sorgere spontanea. Se è vero che non c'è alcuna correlazione fra richieste HTTP, anche se fatte allo stesso server dallo stesso client in rapida successione, e che quindi ogni richiesta/risposta HTTP è una cosa a sè stante, com'è possibile che si riesca a usare HTTP per servizi che richiedono la persistenza di un certo stato? Ad esempio, proviamo a visitare un e-commerce (Amazon, Zalando, etc), ad aggiungere uno o più articoli al carrello e a proseguire la navigazione. Quando navighiamo fra pagine diverse (e quindi facciamo richieste HTTP diverse) notiamo che gli articoli del carrello rimangono in qualche modo memorizzati: c'è quindi **una persistenza dello stato**[8]. Com'è possibile?

Questo è possibile grazie alla capacità dei browser di memorizzare delle informazioni lato client. Avete mai sentito parlare dei **cookie HTTP**? Essi sono un modo per **memorizzare informazioni lato client** ed utilizzarle in una sessione HTTP in maniera da fornire al server delle informazioni come (ad esempio) il numero di articoli presenti in un carrello, o delle preferenze sui nostri indirizzi e-mail in fase di login. I cookie HTTP sono il modo "classico", più vecchio ma più utilizzato, per memorizzare informazioni lato client. In generale è possibile memorizzare delle informazioni sul client in 4 modi[9]:

- Cookie
- Web Storage API: fornisce una sintassi di base per scrivere e leggere semplici oggetti di tipo chiave/valore; usata di solito per memorizzare piccole quantità di dati
- IndexedDB: fornisce al client un vero e proprio database per memorizzare dati più complessi; è possibile anche memorizzare flussi audio o video
- Cache API: è la più moderna libreria per memorizzare dati lato client; questa API è progettata per memorizzare risposte HTTP corrispondenti a specifiche richieste, ed è molto utile per memorizzare asset (risorse statiche come .css, .js, immagini, video, etc) in maniera da non doverle ricaricare dal server a visite successive alla stessa pagina; purtroppo non è supportata da molti browser, il che rende ancora scarso il suo utilizzo.

Tutte queste metodologie sono facilmente utilizzabili grazie al linguaggio Javascript.

Cookie

Un cookie HTTP è un dato (di solito un piccolo dato) che un **server spedisce al browser** dopo una richiesta. Il browser può memorizzare il cookie e rispedirlo indietro alla richiesta successiva allo stesso server[10]. Tipicamente, è usato lato server per capire se due richieste provengono dallo stesso browser (ad esempio per tenere traccia di un utente loggato). È il meccanismo più basilare, quindi, per offrire **persistenza dello stato** in HTTP, un protocollo che per natura è stateless.

I cookie sono usati fondamentalmente per 3 motivi:

- **Gestione della sessione**: login, carrelli negli e-commerce, in generale tutto ciò che il server dovrebbe "ricordarsi" sul client;
- **Personalizzazioni**: preferenze degli utenti, come temi, colori utilizzati su un sito, altre impostazioni
- **Tracking**: Registrare e analizzare il comportamento degli utenti su determinate pagine/siti

Ma come funzionano esattamente i cookie? Quando riceve una richiesta HTTP, un server può spedire al client nella risposta l'header `Set-Cookie`. Il cookie è conservato dal browser, e generalmente rispedito nelle richieste successive a quel server fra con l'header di richiesta HTTP `Cookie`. Può essere specificata dal server una data di scadenza (o una durata massima) del cookie, dopo la quale il cookie non verrà più mandato dal client nelle richieste a quel server. Possono anche essere specificate dal server delle restrizioni ai domini su cui viene usato quel cookie (utile nel caso di siti con svariati domini di terzo livello).

Gli header `Set-Cookie` e `Cookie`

L'header HTTP di risposta Set-Cookie è usato per spedire dati dal server al client. Ecco la sua forma generale di utilizzo:

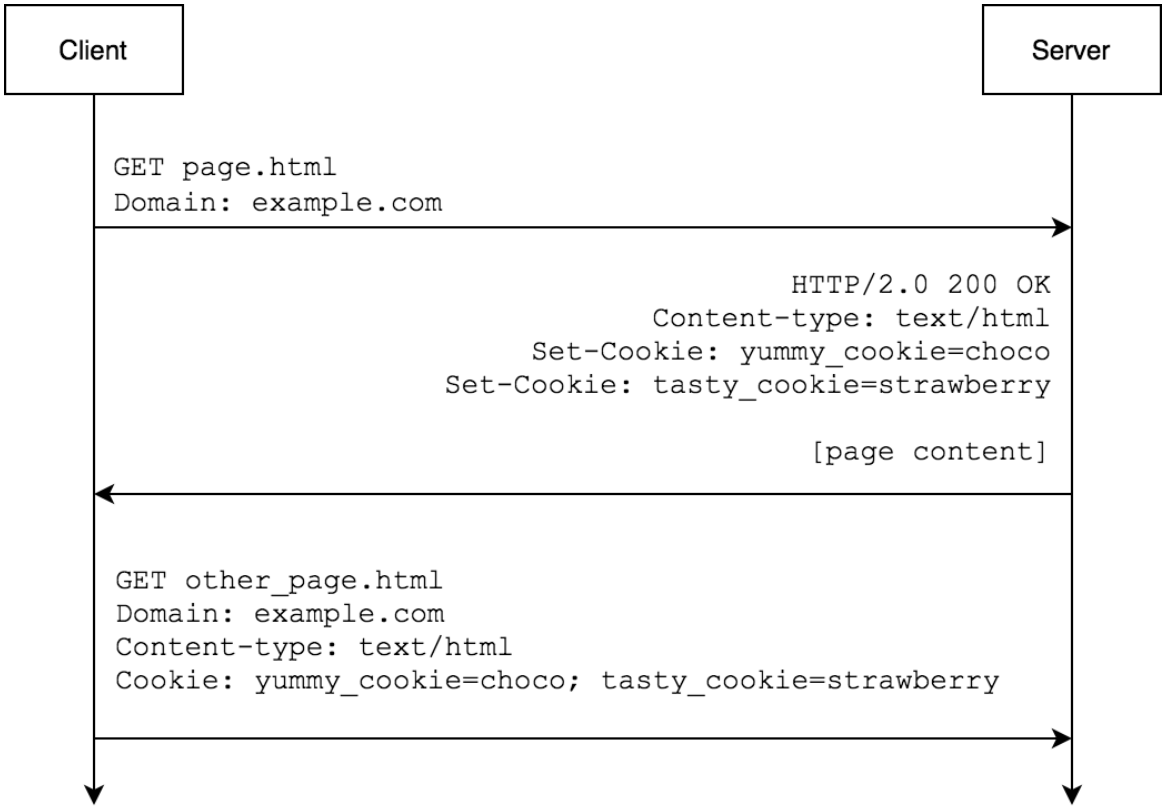
```
Set-Cookie: <cookie-name>=<cookie-value>
```

Settare cookie lato server è piuttosto semplice: a questo link è possibile ad esempio leggere come PHP gestisce i cookie HTTP e come è possibile settare dei cookie in PHP.

Quando un client riceve una risposta dal server contenente questo header memorizza lato client il cookie come indicato dal server. I cookie sono memorizzati dal browser sotto forma di semplice stringa:

```
name-1=val-1;name-2=val-2;...;name-N:val-N
```

Alle richieste successive, un client inserirà nella richiesta l'header di richiesta `Cookie`. Di seguito un esempio grafico di una sessione HTTP con utilizzo di cookie.



Nell'esempio appena visto, il server ha settato un **cookie di sessione**: essi sono cookie validi solo nella sessione corrente che vengono cancellati quando il browser viene chiuso. È possibile per il server creare dei **cookie permanenti**, cioè che rimangono memorizzati nel client anche dopo l'uscita dal programma. Un server setta dei cookie permanenti semplicemente specificando nel Set-Cookie la data di scadenza (Expiration Date) oppure la durata massima (Max-Age):

```
HTTP/2.0 200 OK
...
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT;
Set-Cookie: color=#ff0000; Max-Age=300000;
...
```

La differenza fra `Expires` e `Max-Age` è che il primo indica una data di scadenza esplicita (un oggetto `Date()` Javascript), mentre il secondo indica la durata del cookie in millisecondi. I cookie **permanenti** sono il motivo per cui, una volta loggati a siti come Facebook o instagram, alla visita successiva ai rispettivi siti (anche dopo che il browser è chiuso e il computer spento) ci troviamo ancora loggati.

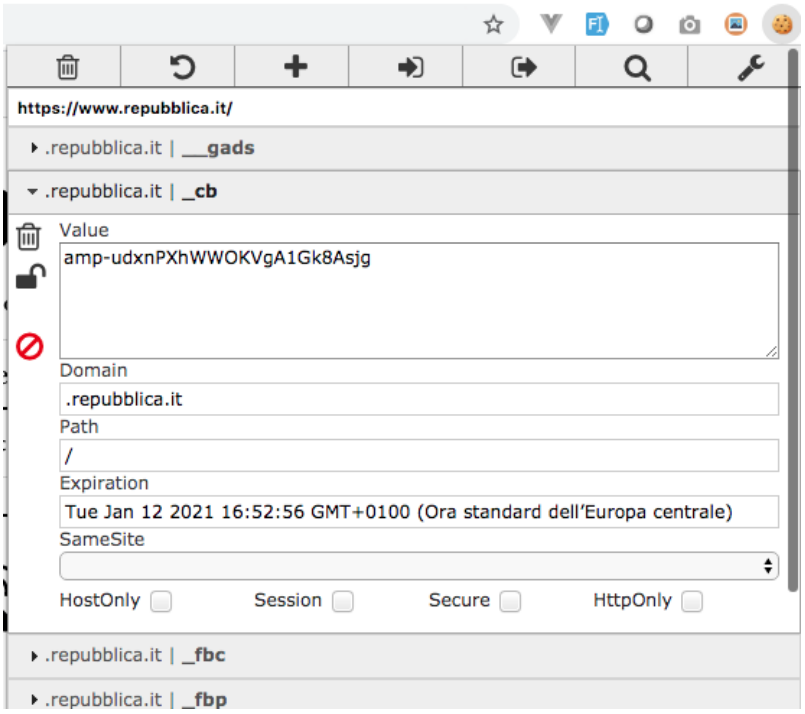
È possibile anche settare un cookie via Javascript. Di seguito alcune semplici funzioni Javascript per leggere, settare cancellare un cookie:

```
function getCookie(name) {
    var v = document.cookie.match('(^|;) ?' + name +
'=([^;]*)(;|$)');
    return v ? v[2] : null;
}

function setCookie(name, value, days) {
    var d = new Date;
    d.setTime(d.getTime() + 24*60*60*1000*days);
    document.cookie = name + "=" + value + ";path=/;expires=" +
d.toGMTString();
}

function deleteCookie(name) {
    setCookie(name, '', -1);
}
```

È possibile anche installare delle estensioni nei browser per visualizzare i cookie memorizzati quando visitiamo determinati siti. Una delle più utilizzate estensioni per Chrome a riguardo si chiama EditThisCookie, disponibile a questo link. Essa aggiunge alla barra superiore di Chrome un pulsante che ha la forma di un biscotto (sì, cookie in inglese significa proprio "biscotto") che una volta cliccato ci fa vedere tutti i cookie memorizzati sul client per il sito che si sta visitando nella scheda corrente. Ecco un esempio di visualizzazione di alcuni cookie presenti sul sito https://www.repubblica.it :



Evoluzione del concetto di risorsa in HTTP

Finora abbiamo detto che HTTP è un protocollo per l'accesso a *risorse* sulla rete Internet, e abbiamo anche parlato di *risorse* statiche, dinamiche e attive; e inoltre abbiamo parlato di URL come di Uniform *Resource* Locator. Abbiamo detto che un metodo HTTP indica un determinato tipo di operazione da fare su una *risorsa*. Insomma, HTTP è strettamente correlato al concetto di risorsa. Ma alla fine, **che cos'è esattamente una risorsa in HTTP?**

Bibliografia/Sitografia

[1] - https://www.ictshore.com/free-ccna-course/application-layer/
[2] - https://www.greenend.org.uk/rjk/tech/smtpreplies.html
[3] - https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
[4] - https://slacksite.com/other/ftp.html
[5] - https://www.w3.org/History/1989/proposal.html
[6] - http://www.nbcnews.com/id/5217598/ns/technology_and_science-tech_and_gadgets/t/web-inventor-finally-makes-money-it/#.XlADJhNKjOQ
[7] - J. Kurose, K. Ross - Reti di calcolatori e Internet. Un approccio top-down (VI edizione - Pearson ed.)
[8] - https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
[9] - https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage
[10] - https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies