

Università degli Studi di Roma "Tor Vergata"
Laurea in Informatica

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2024/2025

Livello di applicazione

dr. Manuel Fiorelli

manuel.fiorelli@uniroma2.it

<https://art.uniroma2.it/fiorelli>

Basate sulle slide del libro di testo:

https://gaia.cs.umass.edu/kurose_ross/ppt.php

Introduction: 1-1

Livello di applicazione: panoramica

- Principi delle applicazioni di rete
- Web e HTTP
- E-mail, SMTP, IMAP
- DNS: il servizio di directory di Internet
- Applicazioni P2P
- Streaming video e reti di distribuzione di contenti
- Programmazione delle socket programming con UDP e TCP



Livello di applicazione: panoramica

Obiettivi:

- Aspetti concettuali e implementativi dei protocolli delle applicazioni di rete
 - modelli di servizio del livello di trasporto
 - paradigma client-server
 - paradigma peer-to-peer
- apprendere informazioni sui protocolli esaminando i protocolli e l'infrastruttura delle più diffuse applicazioni di rete
 - HTTP
 - SMTP, IMAP
 - DNS
 - Sistemi per lo streaming video, CDNs
- programmare applicazioni di rete
 - socket API

Alcune diffuse applicazioni di rete

- posta elettronica (e-mail)
- trasferimento file
- newsgroup (es. Usenet) e poi forum
- Web
- messaggistica istantanea
- social media
- giochi multiutente via rete
- streaming di video-clip memorizzati (YouTube, Hulu, Netflix)
- condivisione di file P2P
- Applicazioni basate sulla posizione
- Applicazioni di pagamento mobile
- telefonia via Internet, voice-over-IP (VoIP), es. Skype
- videoconferenza in tempo reale
- ricerca sul Web
- shell o desktop remoto
- ...

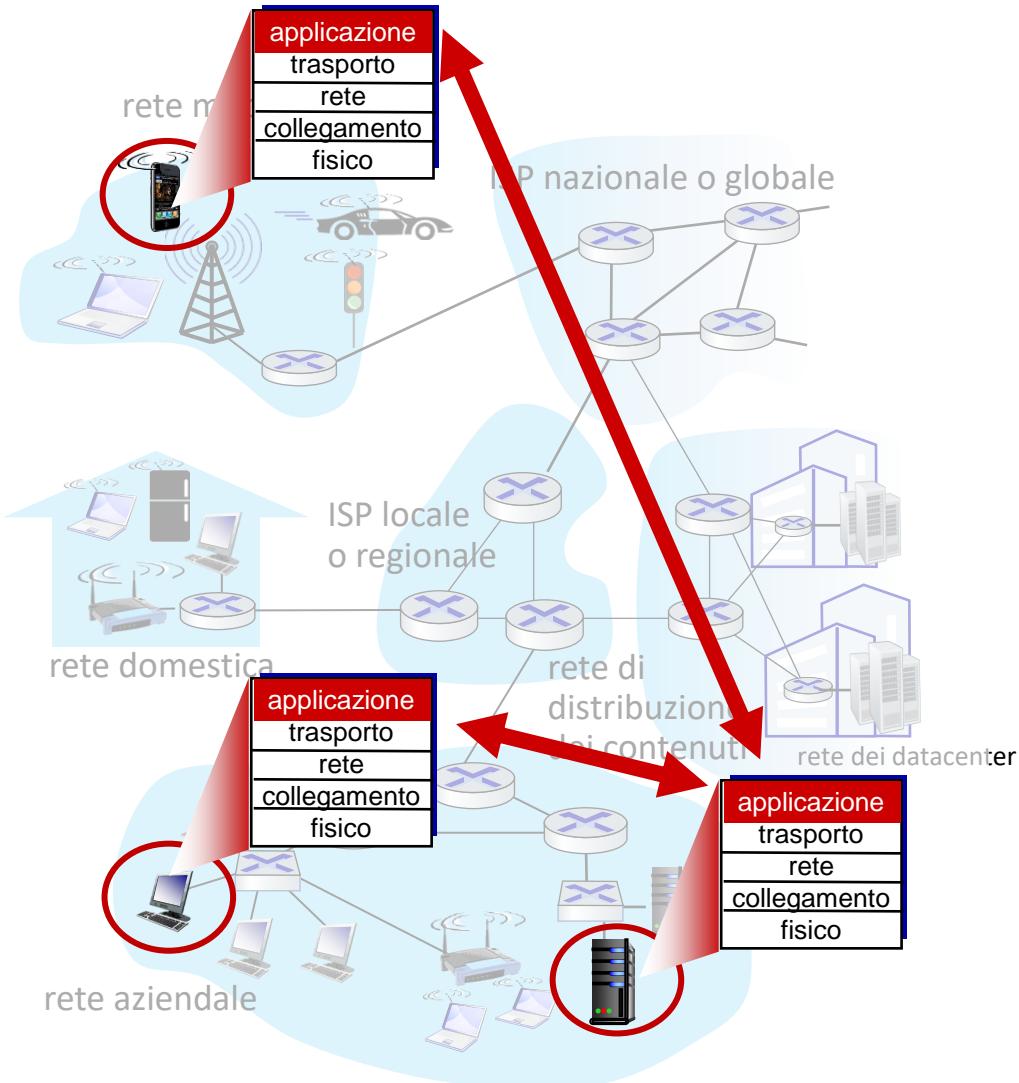
Creare un'applicazione di rete

scrivere programmi che:

- girano su sistemi periferici differenti
- comunicano attraverso la rete
- ad esempio, il software di un server Web comunica con il software di un browser

non è necessario scrivere programmi per i dispositivi nel nucleo della rete

- i dispositivi del nucleo della rete non eseguono applicazioni utente
- il confinamento delle applicazioni nei sistemi periferici ha facilitato il rapido sviluppo e la diffusione di una vasta gamma di applicazioni per Internet



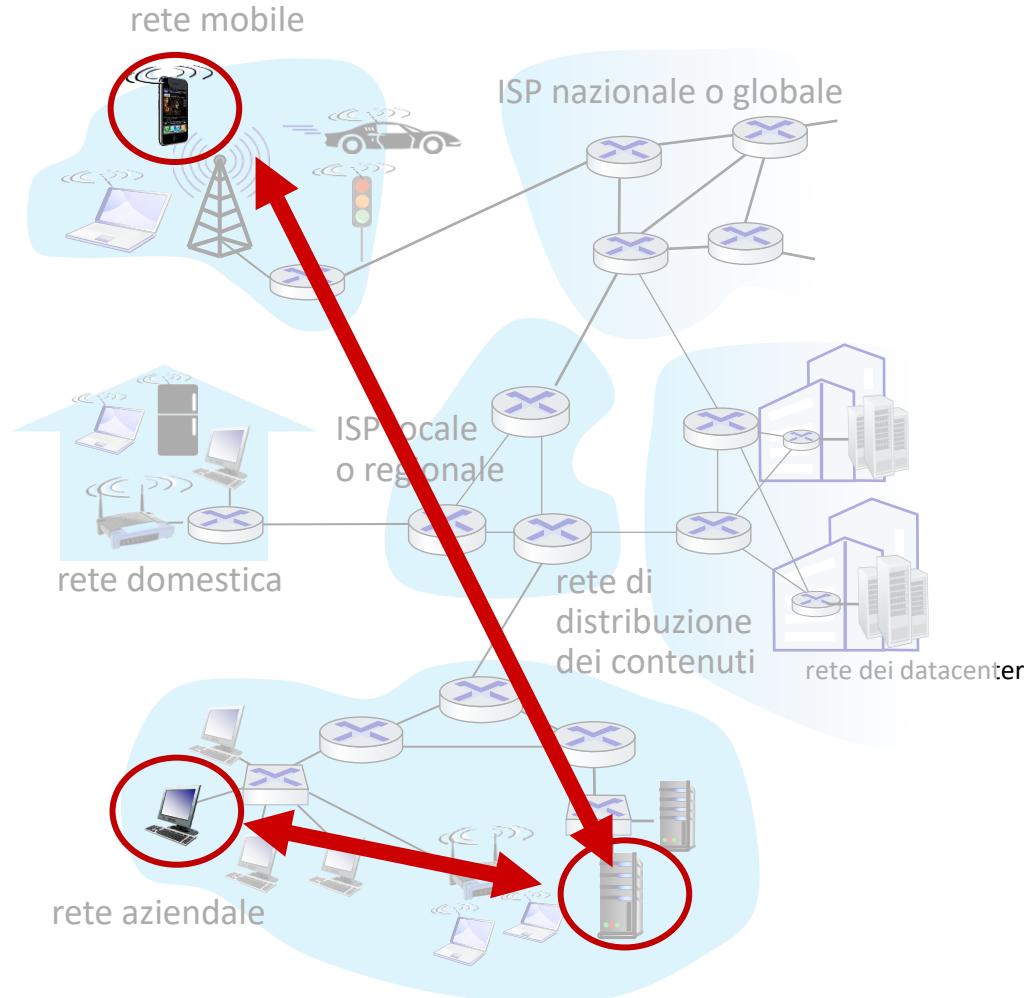
Paradigma client-server

server:

- host sempre attivo
- indirizzo IP fisso
- spesso in datacenter, per la scalabilità
- onere economico sui fornitori del servizio (per infrastruttura, manutenzione, traffico)

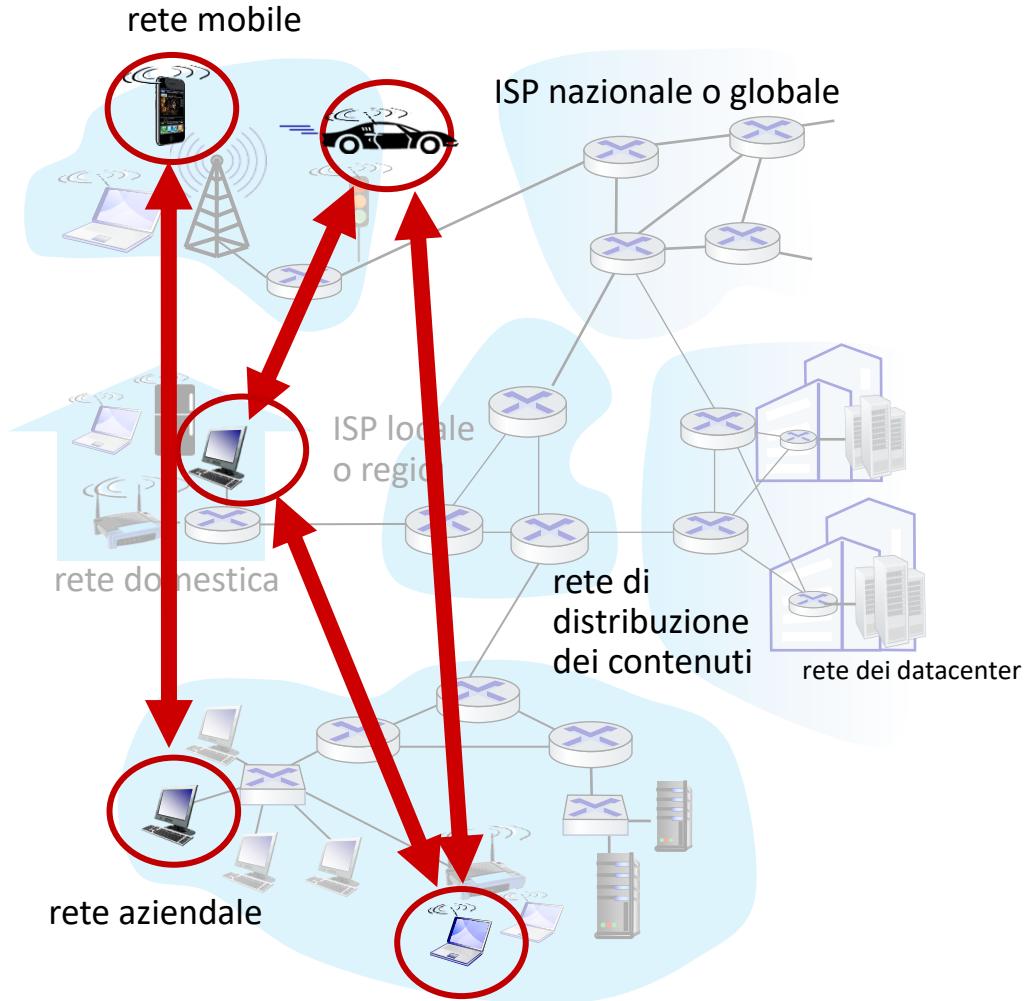
client:

- contatta, comunica col server
- può contattare il server in qualunque momento
- può avere indirizzi IP dinamici
- *non* comunica direttamente con gli altri client
- esempi: Web, posta elettronica, etc.



Architettura peer-to-peer

- *non c'è un server sempre attivo*
- coppie arbitrarie di host (peer) comunicano direttamente tra loro
- i peer richiedono un servizio ad altri peer e forniscono un servizio in cambio ad altri peer.
 - *scalabilità intrinseca* – nuovi peer aggiungono capacità di servizio al sistema, sebbene generino anche nuovo carico di lavoro
 - onere economico ridotto sui fornitori del servizio
- i peer non devono necessariamente essere sempre attivi e cambiano indirizzo IP
 - Difficile da gestire
- esempio: condivisione di file P2P [BitTorrent]



Processi comunicanti

processo: programma in esecuzione su di un host

- all'interno dello stesso host, due processi comunicano usando un approccio interprocesso (*inter-process communication*) (definito dal sistema operativo)
- processi su host differenti comunicano attraverso lo scambio di *messaggi* usando il servizio del livello di trasporto

client, server

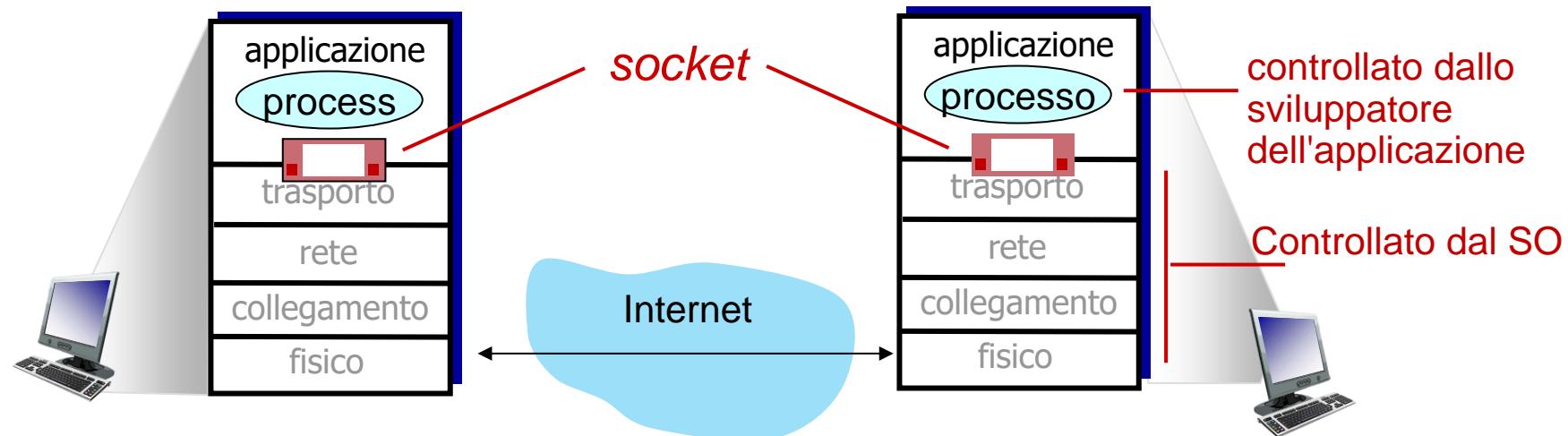
processo client: processo che dà inizio alla comunicazione

processo server: processo che attende di essere contattato

- nota: nelle applicazioni P2P, un processo può essere sia client sia server; cionondimeno, nel contesto di una specifica sessione di comunicazione, ogni processo adotterà l'uno o l'altro ruolo

Socket

- un processo invia/riceve messaggi a/di la sua **socket**
- una socket è analoga a una porta
 - Il processo mittente fa uscire il messaggio fuori dalla propria "porta" (socket)
 - Il processo mittente presuppone l'esistenza di un'infrastruttura esterna che trasporterà il messaggio attraverso la rete fino alla "porta" del processo di destinazione



Indirizzamento

- per ricevere messaggi, un processo deve avere un *identificatore*
- Un host ha un indirizzo IP univoco a 32 bit (IPv4) o 128 bit (IPv6)
- D: è sufficiente conoscere l'indirizzo IP dell'host su cui è in esecuzione un process per identificare il processo stesso?
- R: no, sullo stesso host possono essere in esecuzione *multi* processi.
- *L'identificatore* comprende sia l'indirizzo IP (IP address) che i numeri di porta (port number) associati al processo in esecuzione su un host.
- numeri di porta assegnati (dallo IANA) a applicazioni più note:
 - HTTP server: 80
 - mail server: 25
- Per inviare un messaggio HTTP al server gaia.cs.umass.edu :
 - **indirizzo IP**: 128.119.245.12
 - **numero di porta**: 80

Un protocollo a livello applicazione definisce

- tipi di messaggi scambiati,
 - es, richiesta, risposta
- sintassi dei messaggi:
 - quali sono i campi nel messaggio e come sono descritti
- semantica dei messaggi
 - significato delle informazioni nei campi
- regole per determinare quando e come un processo invia e risponde ai messaggi

protocolli di pubblico dominio:

- definiti nelle RFC, ognuno ha accesso alla definizione
- consentono l'interoperabilità
- ad esempio, HTTP, SMTP

protocolli proprietari:

- ad esempio, Skype, Zoom

Quale servizio di trasposto richiede un'applicazione?

perdita di dati

- alcune applicazioni (es: trasferimento file, transazioni web) richiedono un trasferimento 100% affidabile
- altre applicazioni (es: applicazioni multimediali audio/video) possono tollerare qualche perdita

sensibilità al fattore tempo

- alcune applicazioni (ad esempio, telefonia via Internet, giochi interattivi) richiedono che i ritardi siano bassi per essere “efficaci”

throughput

- alcune applicazioni (dette "sensibili alla banda"), ad esempio, quelle multimediali, per essere efficaci richiedono un'ampiezza di banda minima
- altre applicazioni (le “applicazioni elastiche”) utilizzano l'ampiezza di banda che si rende disponibile

sicurezza

- riservatezza, integrità dei dati, autenticazione, ...

Requisiti del servizio di trasporto di alcune applicazioni comuni

<u>applicazione</u>	tolleranza alla perdita di dati	throughput	sensibilità al fattore tempo
trasferimento file	no	variabile	no
posta elettronica	no	variabile	no
documenti Web	no	variabile	no
audio/video in tempo reale	sì	audio: 5kbps-1Mbps video:10kbps-5Mbps	sì, centinaia di ms
streaming audio/video memorizzati	Sì	come sopra	sì, pochi secondi
giochi interattivi	sì	fino a pochi kbps	sì, centinaia di ms
messaggistica istantanea	no	variabile	sì e no

Servizi dei protocolli di trasporto di Internet

Servizio di TCP:

- **trasporto affidabile** fra i processi di invio e di ricezione: dati consegnati senza errori, perdite e nell'ordine di invio
- **controllo di flusso**: il mittente non vuole sovraccaricare il destinatario
- **controllo della congestione**: "strozza" il processo d'invio quando la rete è sovraccarica
- **orientato alla connessione**: è richiesto un setup fra i processi client e server
- **non offre**: temporizzazione, garanzie su un'ampiezza di banda minima, sicurezza

Servizio di UDP:

- **trasferimento dai inaffidabile** fra i processi d'invio e di ricezione
- **non offre**: affidabilità, controllo di flusso, controllo della congestione, temporizzazione, ampiezza di banda minima, sicurezza, né setup della connessione.

D: perché preoccuparsi?
Perché c'è UDP?

R: UDP offre alle applicazioni maggiore controllo sull'invio dei dati (senza garanzie sul throughput end-to-end disponibile, né ritardi)

Applicazioni Internet: protocollo a livello applicazione e protocollo di trasporto

applicazione	protocollo a livello applicazione	Protocollo di trasporto sottostante
trasferimento file	FTP [RFC 959]	TCP
posta elettronica	SMTP [RFC 5321]	TCP
documenti web	HTTP [RFC 7230, 9110]	TCP
telefonia via Internet	SIP [RFC 3261], RTP [RFC 3550], o proprietario	TCP o UDP
streaming audio/video	HTTP [RFC 7230], DASH	TCP
giochi interattivi	WOW, FPS (proprietario)	UDP o TCP

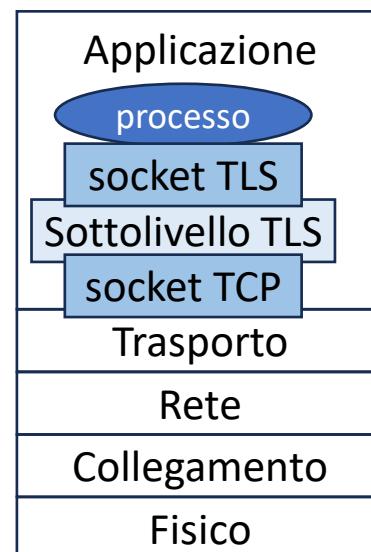
Rendere sicuro TCP

Socket TCP e UDP:

- nessuna cifratura
- password inviate in chiaro, cioè senza cifratura, attraverso socket attraversano Internet in chiaro(!)

Transport Layer Security (TLS)

- offre connessioni TCP cifrate
- controllo di integrità dei dati
- autenticazione end-to-end



TLS implementato a livello applicazione

- le applicazioni usano librerie TLS, che usano a propria volta TCP
- testo in chiaro (*cleartext*) inviato nella “socket TLS” viene *cifrato* prima di essere immesso nella socket TCP e attraversare Internet fino a destinazione; qui, i dati (cifrati) consegnati dalla socket TCP sono decifrati prima di essere restituiti in chiaro dalla socket TLS

Livello applicazione: panoramica

- Principi delle applicazioni di rete
- **Web e HTTP**
- E-mail, SMTP, IMAP
- DNS: il servizio di directory di Internet
- Applicazioni P2P
- Streaming video e reti di distribuzione di contenti
- Programmazione delle socket programming con UDP e TCP



Web e HTTP

*gli oggetti sono in realtà chiamati *risorse*

**lo *schema* indica il protocollo usato per accedere alla risorsa; altri esempi, sono *https* per HTTP sopra TLS o *ftp* per l'accesso tramite FTP

Ripassiamo la terminologia...

- una pagina web è costituita da *oggetti**, ciascuno dei quali può essere memorizzato su un server Web differente
- un oggetto può essere un file HTML, un'immagine JPEG, uno script JavaScript, un foglio di stile CSS, un file audio, ...
- una pagina web è formata da *un file HTML di base* che include *diversi oggetti referenziati, ciascuno* referenziato da un *URL*, ad esempio,

http://www.someschool.edu/someDept/pic.gif

schema**
(scheme)

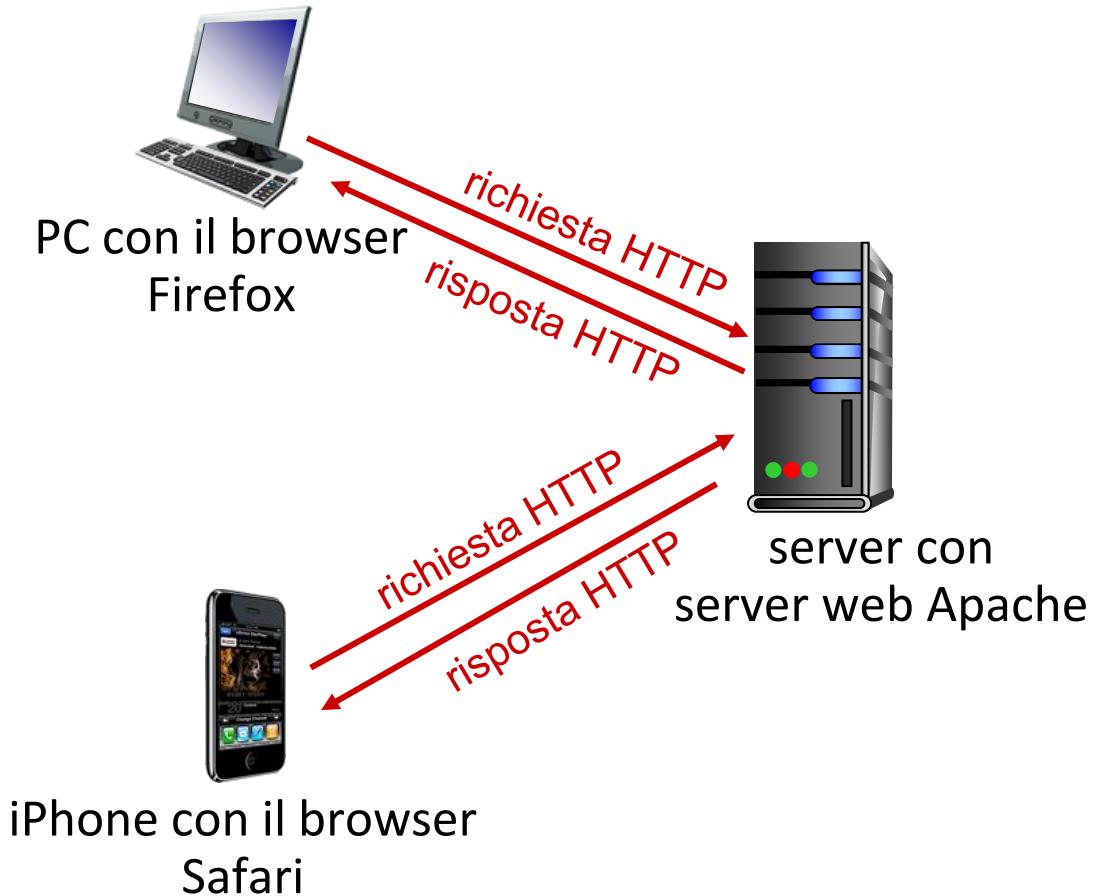
nome dell'host
(host name)

percorso
(path name)

Panoramica su HTTP

HTTP: HyperText Transfer Protocol

- protocollo a livello applicazione del Web
- modello client/server:
 - *client*: browser che richiede, riceve (usando il protocollo HTTP) e “visualizza” gli oggetti del Web
 - *server*: il server Web che invia (usando il protocollo HTTP) oggetti in risposta alle richieste



Panoramica su HTTP (continua)

HTTP usa TCP:

- il client inizializza la connessione TCP (crea una socket) con il server, sulla porta 80
- il server accetta la connessione TCP dal client
- messaggi HTTP (messaggi di un protocollo di applicazione) scambiati tra browser (client HTTP) e server Web (server HTTP)
- connessione chiusa TCP

HTTP è un protocollo “senza stato” (stateless)

- Il server non mantiene informazioni sulle richieste fatte dal client

nota

i protocolli che mantengono lo stato sono complessi!

- la storia passato (stato) deve essere memorizzata
- se il server e/o il client si bloccano, le loro viste dello "stato" potrebbero essere contrastanti e dovrebbero essere riconciliate

Connessioni HTTP: due tipi

Connessioni non persistenti

1. connessione TCP aperta
2. una singola coppia richiesta/risposta HTTP per un singolo oggetto
3. connessione TCP chiusa

Lo scaricamento di oggetti multipli richiede connessioni multiple

Connessioni persistenti

- connessione TCP connection al server aperta
- più coppie richiesta/risposta HTTP, per trasmettere *più oggetti* su una *singola connessione* TCP tra client e server
- connessione TCP chiusa

Connessioni non persistenti

L'utente immette l'URL: `http://www.someSchool.edu/someDepartment/home.html`
(contiene testo, riferimenti a 10 immagini JPEG)



tempo ↓

1a. Il client HTTP client inizializza una connessione TCP con il server HTTP (processo) a www.someSchool.edu sulla porta 80

2. Il client HTTP trasmette *un messaggio di richiesta* HTTP (contenente l'URL) nella socket della connessione TCP. Il messaggio indica che il client vuole l'oggetto someDepartment/home.html



1b. Il server HTTP all'host www.someSchool.edu in attesa di una connessione TCP alla porta 80 "accetta" la connessione e avvisa il client

3. Il server HTTP riceve il messaggio di richiesta, forma il *messaggio di risposta* che contiene l'oggetto richiesto e invia il messaggio nella sua socket

Connessioni non persistenti (continua)

L'utente immette l'URL: `http://www.someSchool.edu/someDepartment/home.html`
(contiene testo, riferimenti a 10 immagini JPEG)



- time ↓
5. Il client HTTP riceve il messaggio di risposta che contiene il file HTML e visualizza il document HTML.
Esamina il file HTML, trova I riferimenti a 10 oggetti JPEG
 6. I passi 1-5 sono ripetuti per ciascuno dei 10 oggetti JPEG
4. Il server HTTP chiude la connessione TCP
- 

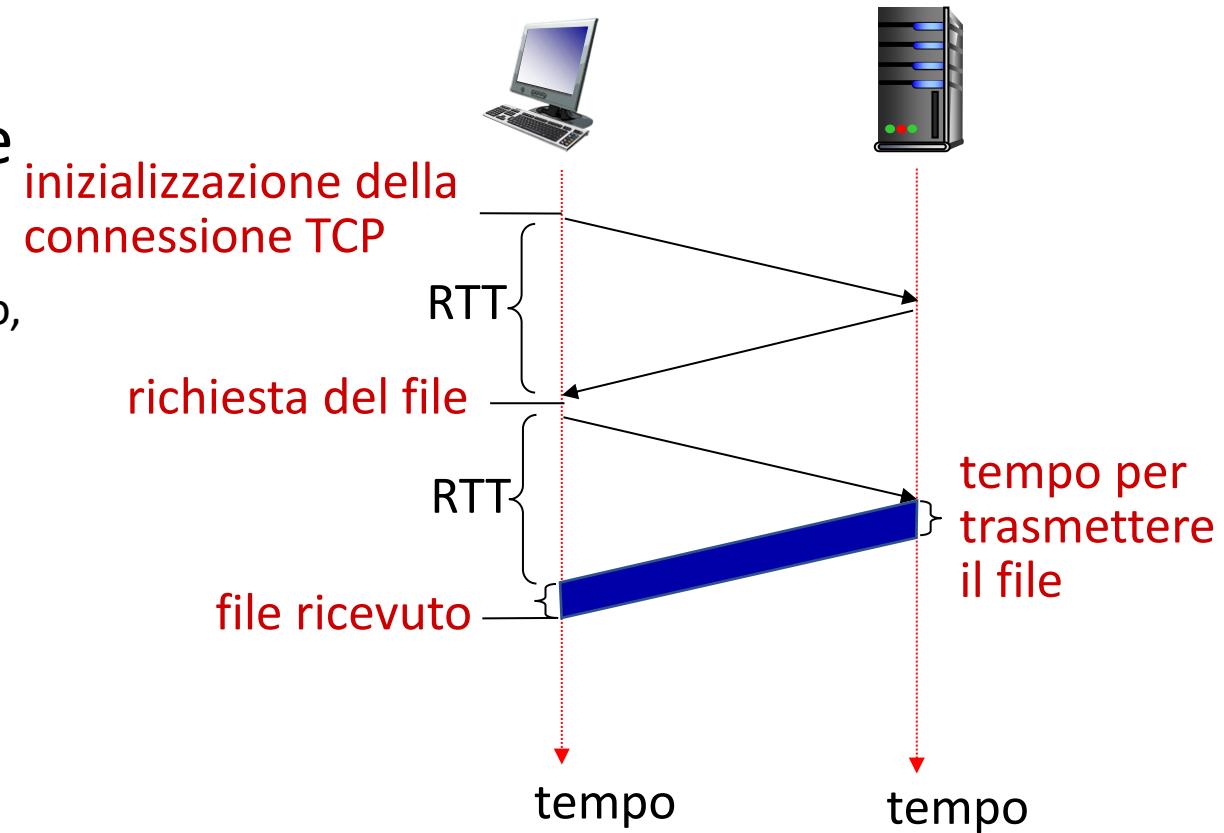
Connessioni non persistenti: tempo di risposta

RTT (definizione): tempo impiegato da un piccolo pacchetto per andare dal client al server e ritornare al client (include ritardi di elaborazione, accodamento, propagazione)

Tempo di risposta (per oggetto):

- un RTT per inizializzare la connessione TCP
- un RTT perché ritornino la richiesta HTTP e i primi byte della risposta HTTP
- tempo di trasmissione del file/oggetto

$$\text{Tempo di risposta con connessioni non persistenti} = 2\text{RTT} + \text{tempo di trasmissione del file}$$



Connessioni persistenti (HTTP 1.1)

Svantaggi delle connessioni non persistenti:

- richiedono 2 RTT per oggetto
- allocazione di buffer e variabili per *ogni* connessione TCP
- i browser spesso aprono connessioni TCP parallele per caricare gli oggetti referenziati
 - maggiore onere sul server web

Connessioni persistenti (HTTP1.1):

- il server lascia la connessione TCP aperta dopo l'invio di una risposta
- i successivi messaggi tra gli stessi client/server vengono trasmessi sulla connessione aperta
- il client invia le richieste non appena incontra un oggetto referenziato
- un solo RTT per tutti gli oggetti referenziati

Messaggio di richiesta HTTP

- Due tipi di messaggi HTTP: *richiesta, risposta*
- **Messaggio di richiesta HTTP:**
 - ASCII (formato leggibile dall'utente)

riga di richiesta (*request line*)

(comandi GET, POST,
HEAD)

carattere di ritorno a capo (*carriage return*)
carattere di nuova linea (*line-feed*)

Un carriage return e un, 
line feed all'inizio della
linea indicano la fine delle
righe di intestazione

* Check out the online interactive exercises for more
examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Messaggio di richiesta HTTP (continua)

Alcuni campi di intestazione nei messaggi di richiesta:

Host

- hostname e numero di porta (se assente, si assume 80 per HTTP e 443 per HTTPS) del server al quale sarà inviata la richiesta. Obbligatorio in HTTP/1.1; se assente, il server può rispondere con un 400 Bad Request. Necessario per il funzionamento delle web cache e del name-based virtual hosting.

User-Agent

- Identifica l'applicazione, il sistema operativo, il *vendor* e/o la versione dello *user agent* che sta effettuando la richiesta

Accept

- Tipi di contenuto, espressi come *media type*, compresi dal client

Accept-Language

- Linguaggi naturali o *locale* preferiti dal client

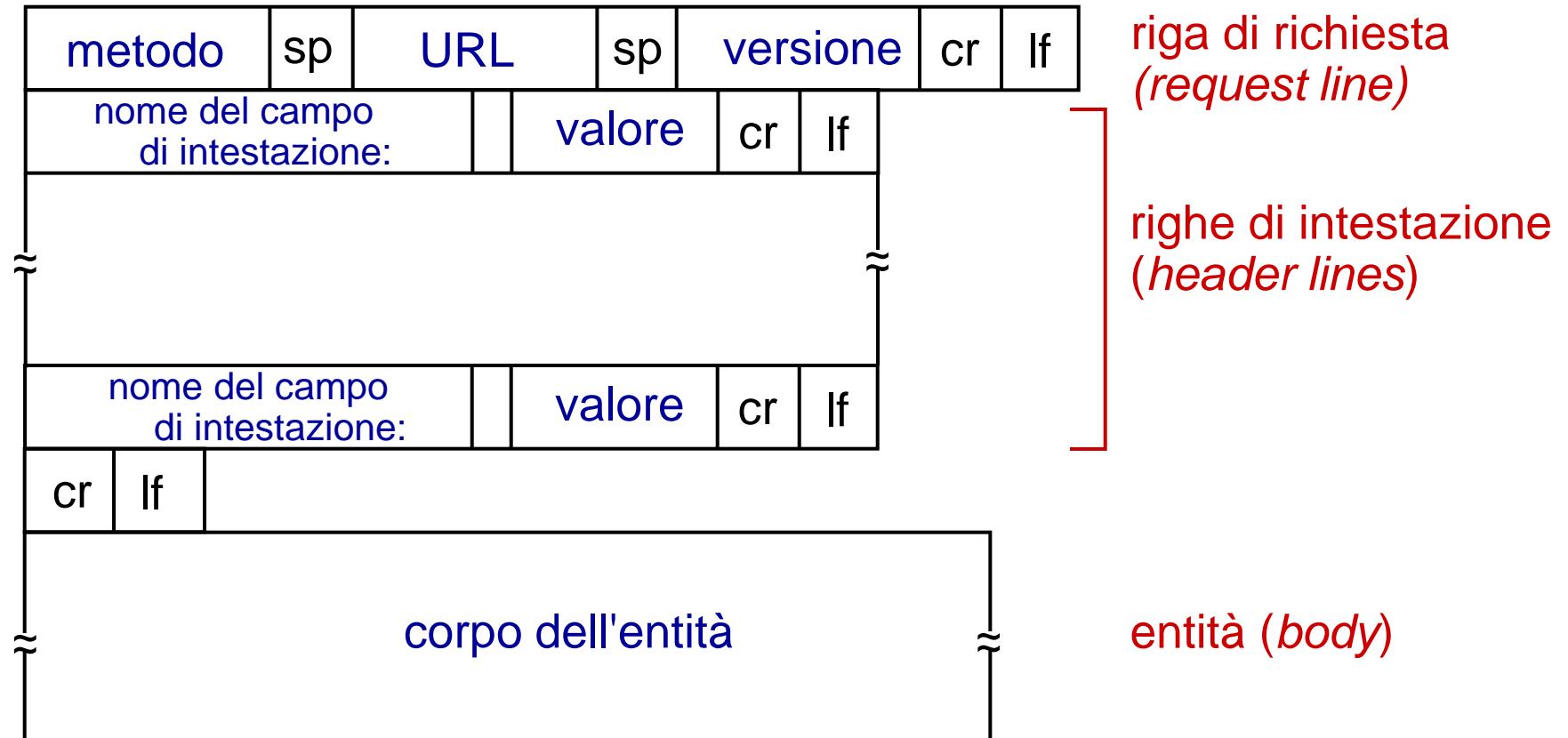
Accept-Encoding

- Algoritmi di codifica (tipicamente, la compressione) compresi dal client

Connection

- Controlla se la connessione rimarrà aperta al termine dello scambio richiesta/risposta. Il valore *close* indica che la connessione sarà chiusa (default in HTTP/1.0); altrimenti, una lista non vuota di nomi di header (in genere solo *keep-alive*), che saranno rimossi dal primo proxy non trasparente o cache, indica che la connessione rimarrà aperta (default in HTTP/1.1)

Messaggio di richiesta HTTP: formato generale



Altri messaggi di richiesta HTTP

Metodo POST:

- la pagina web spesso include un form per l'input dell'utente
- l'input dell'utente viene inviato dal client al server nel corpo dell'entità di un messaggio di richiesta HTTP POST

Metodo GET (per inviare dati al server):

- l'input arriva al server nel campo URL della riga di richiesta (dopo un '?'):

```
http://www.somesite.com  
/search?q=monkey&kingdom=animals
```

Metodo HEAD:

- richiede le intestazioni (solo) che verrebbero restituite se l'URL specificato fosse richiesto con il metodo HTTP GET.

Metodo PUT:

- carica un nuovo file (oggetto) sul server
- sostituisce completamente il file esistente all'URL specificato con il contenuto del corpo dell'entità del messaggio di richiesta HTTP PUT.

HTTP: Idempotenza

Un'operazione si dice *idempotente* se l'effetto inteso sul server di una singola richiesta è lo stesso di quello di più richieste identiche

- i metodi GET, PUT, HEAD sono idempotenti
- il metodo POST non è idempotente (quantomeno non in generale)

Un form può essere implementato usando:

- GET: se l'operazione associata al form è idempotente (es. ricerca)
- POST: se l'operazione associata al form NON è idempotente (es. aggiunta di un prodotto al carrello di un e-commerce)

Il concetto di idempotenza è utile per stabilire se un client può *ritentare* in automatico una richiesta in caso di problemi di rete (transitori).

Messaggio di risposta HTTP

riga di stato (*status line*)

HTTP/1.1 200 OK

The diagram illustrates the structure of an HTTP status line. It consists of three parts: 'HTTP/1.1' (version), '200' (status code), and 'OK' (status text). Three blue arrows point from the labels on the left to these components. The first arrow points to 'HTTP/1.1' with the label 'versione del protocollo (protocol version)'. The second arrow points to '200' with the label 'codice di stato (status code)'. The third arrow points to 'OK' with the label 'espressione di stato (status text)'.

- versione del protocollo (*protocol version*)
- codice di stato (*status code*)
- espressione di stato (*status text*)

Messaggio di risposta HTTP (continua)

Alcuni campi di intestazione nei messaggi di richiesta:

Date

- la data e l'ora in cui il messaggio è stato originato

Server

- descrive il software usato dal server di origine per gestire la richiesta (nota: troppi dettagli possono aiutare i malintenzionati a attaccare il server)

Last-Modified

- la data e l'ora in cui il server di origine crede che l'oggetto sia stato modificato per l'ultima volta

Accept-Ranges

- Indica il supporto del server ai download parziali: il valore, se diverso da none, indica l'unità che si può usare per esprimere l'intervallo richiesto

Content-Length

- lunghezza in byte del corpo dell'entità inviato al ricevente (o che *sarebbe* stato inviato nel caso di un richiesta HEAD)

Content-Type

- *media type* (che indica un formato) del corpo dell'entità inviato al ricevente (o che *sarebbe* stato inviato nel caso di un richiesta HEAD)

Codici di stato della risposta HTTP

- Nella prima riga nel messaggio di risposta dal server al client
- Definiti da RFC 9110
- Raggruppati in cinque categorie, discriminate dalla prima cifra

1xx Informational

- una risposta intermedia per comunicare lo stato di connessione o l'avanzamento della richiesta prima di completare l'azione richiesta e inviare una risposta finale (assenti in HTTP/1.0)

2xx Successful

- La richiesta è stata ricevuta con successo, compresa e accettata

3xx Redirect

- Il client deve eseguire ulteriori azioni per soddisfare la richiesta

4xx Client Error

- La richiesta è sintatticamente scorretta o non può essere soddisfatta

5xx Server Error

- Il server ha fallito nel soddisfare una richiesta apparentemente valida

Codici di stato della risposta HTTP

- Nella prima riga nel messaggio di risposta dal server al client
- Alcuni codici di stato e relative espressioni

200 OK

- La richiesta ha avuto successo; l'oggetto richiesto viene inviato nella risposta

301 Moved Permanently

- L'oggetto richiesto è stato trasferito; la nuova posizione è specificata nell'intestazione Location: della risposta

400 Bad Request

- Il messaggio di richiesta non è stato compreso dal server

404 Not Found

- Il documento richiesto non si trova su questo server

406 Not Acceptable

- L'oggetto richiesto non esiste in una forma che soddisfa i vari Accept-*

505 HTTP Version Not Supported

- Il server non ha la versione di protocollo HTTP

Provatate HTTP (lato client)

1. Collegatevi via netcat al vostro server web preferito:

```
% nc -c -v gaia.cs.umass.edu 80 (for Mac)
```

```
>ncat -C gaia.cs.umass.edu 80 (for Windows)
```

- apre una connessione TCP alla porta 80 (porta di default per un server HTTP) dell'host gaia.cs.umass.edu.
- tutto ciò che digitate viene trasmesso alla porta 80 di gaia.cs.umass.edu

2. type in a GET HTTP request:

**GET /kurose_ross/interactive/index.php HTTP/1.1
Host: gaia.cs.umass.edu**

- digitando questo (premete due volte il tasto Invio), trasmettete una richiesta GET minima (ma completa) al server HTTP

3. guardate il messaggio di risposta trasmesso dal server HTTP!

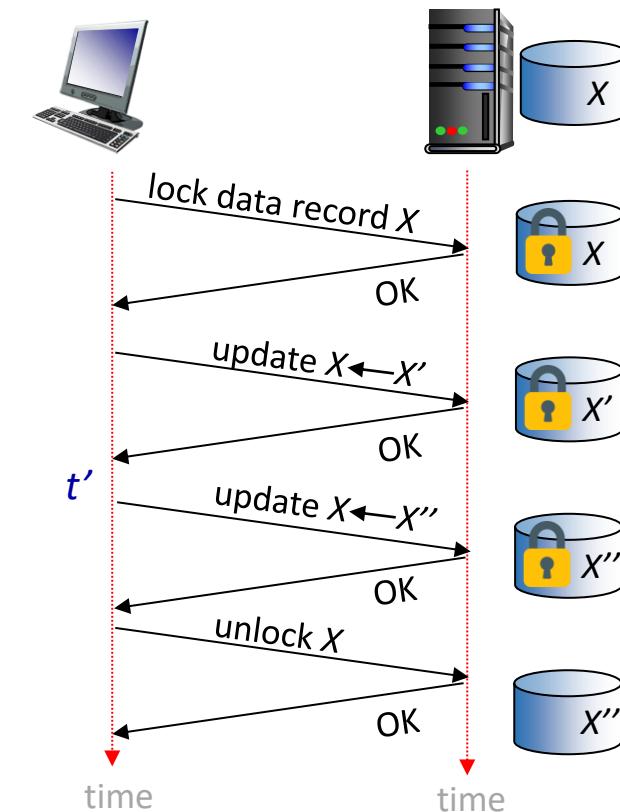
(oppure usate Wireshark per guardare la richiesta e la risposta HTTP catturate)

Mantenere stato utente/server: i cookie

Ricordate: l'interazione HTTP
GET/risposta è *senza stato*
(stateless)

- nessuna nozione di scambio di messaggi HTTP in più fasi per completare una "transazione" Web
 - non è necessario che il client o il server tengano traccia dello "stato" dello scambio in più fasi
 - tutte le richieste HTTP sono indipendenti l'una dall'altra
 - non è necessario che il client né il server siano in grado di "recuperare" da una transazione quasi completa ma mai completata

un protocollo con stato: il client fa due modifiche a X, o nessuna



Q: che succede se la connessione di rete o il client si blocca al tempo t' ?

Mantenere stato utente/server: i cookie

I siti web e il browser client usano i **cookie** per mantenere dello stato tra le transazioni

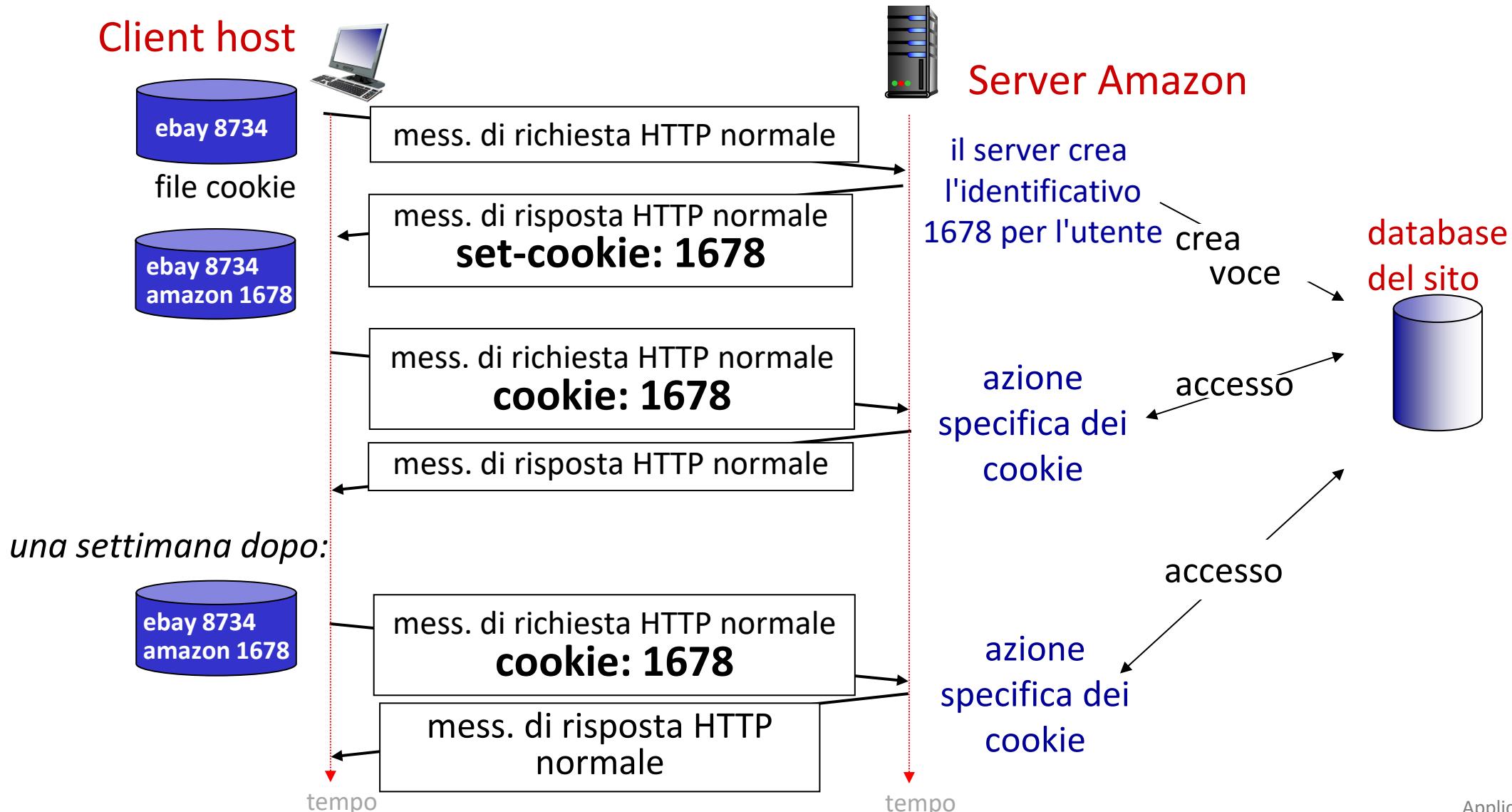
quattro componenti:

- 1) una riga di intestazione nel messaggio di *risposta* HTTP
- 2) una riga di intestazione nel messaggio di *richiesta* HTTP
- 3) un file cookie mantenuto sul sistema terminale dell'utente e gestito dal browser dell'utente
- 4) un database sul sito

Esempio:

- Susan usa il browser dal portatile, visita uno specifico siti di commercio elettronico per la prima volta
- quando la richiesta HTTP iniziale arriva al sito, il sito crea:
 - un identificativo unico
 - una voce nel proprio database, indicizzata dal numero identificativo
- Il server ritorna una risposta che include l'intestazione Set-cookie, che contiene l'identificativo unico e che sarà aggiunto al file dei cookie
- le successive richieste del browser di Susan per questo sito conterranno l'identificativo in una intestazione cookie

Mantenere stato utente/server: i cookie



Cookie HTTP: commenti

I cookie possono essere usati per:

- autorizzazione
- carrello degli acquisti
- raccomandazioni
- stato della sessione dell'utente (e-mail Web)

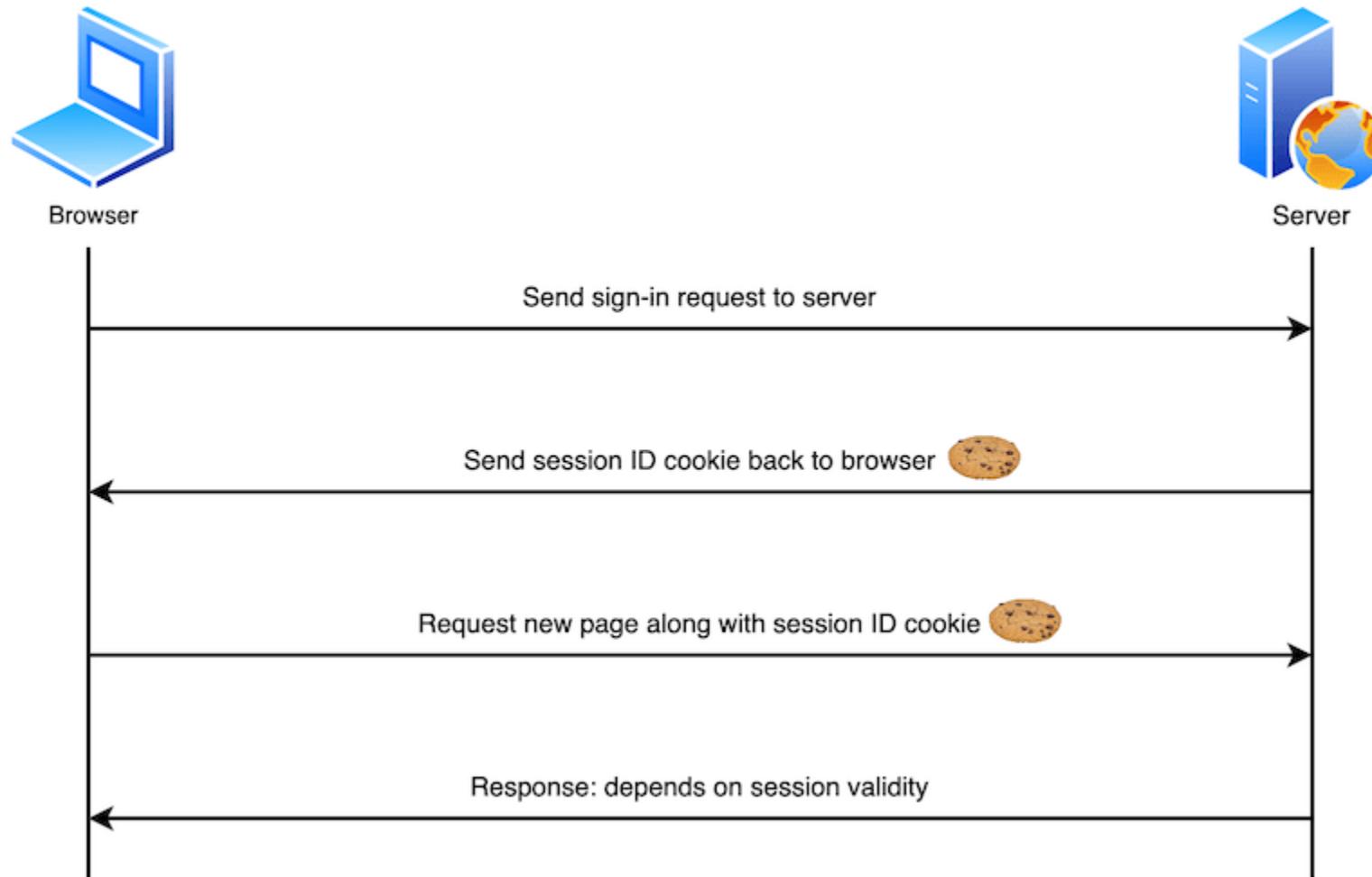
Sfida: Come mantenere lo stato?

- *presso gli endpoint del protocollo:* mantenere lo stato presso il trasmittitore e il ricevitore attraverso multiple transazioni
- *nei messaggi:* i cookie trasportano lo stato nei messaggi HTTP

nota
cookie e privacy:

- I cookie consentono ai siti di *imparare* molto di voi
- cookie persistenti di terze parti (cookie di tracciamento, *tracking cookies*) consentono il tracciamento di una identità comune (valore del cookie) attraverso siti web multipli

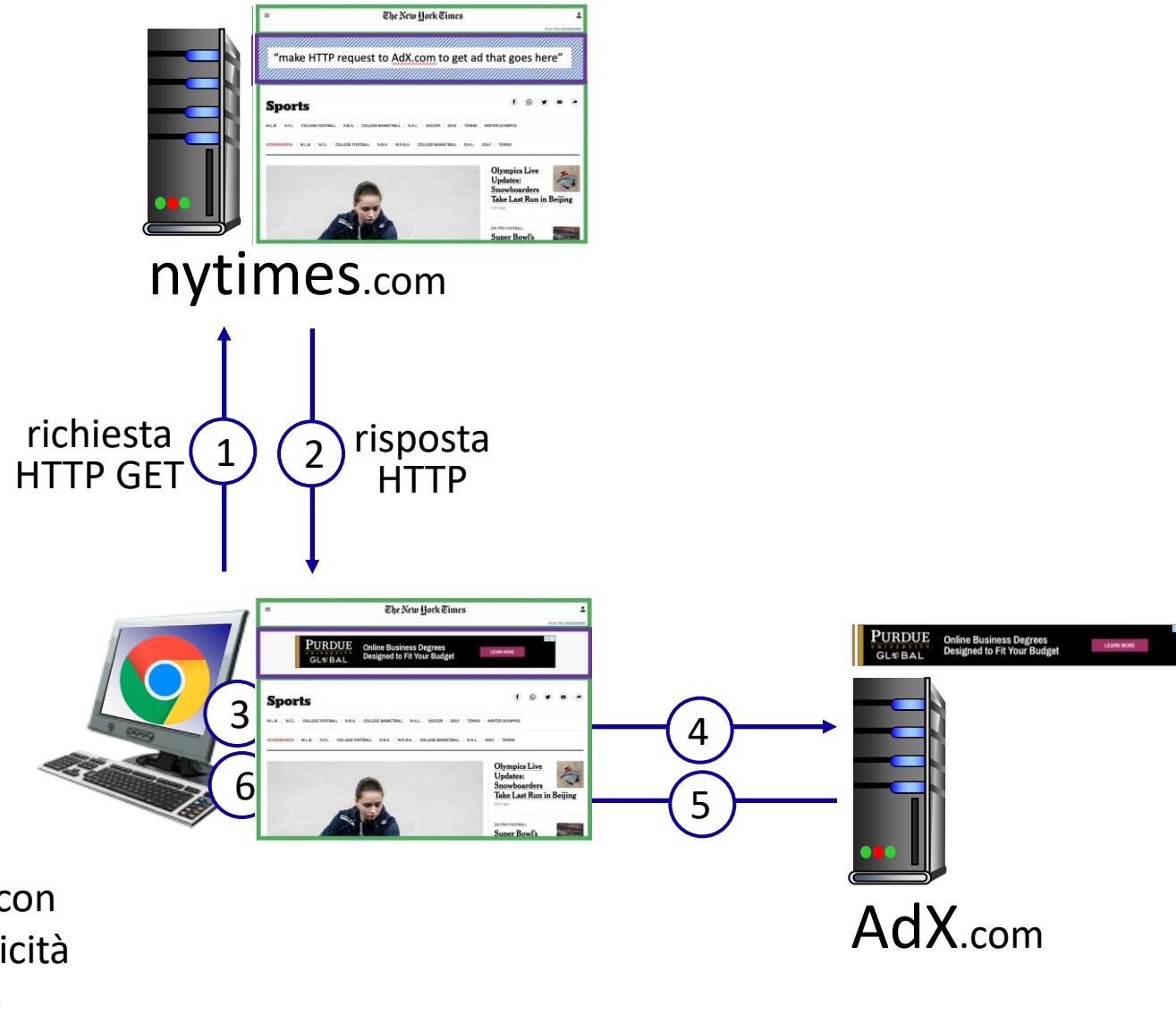
Esempio: autorizzazione



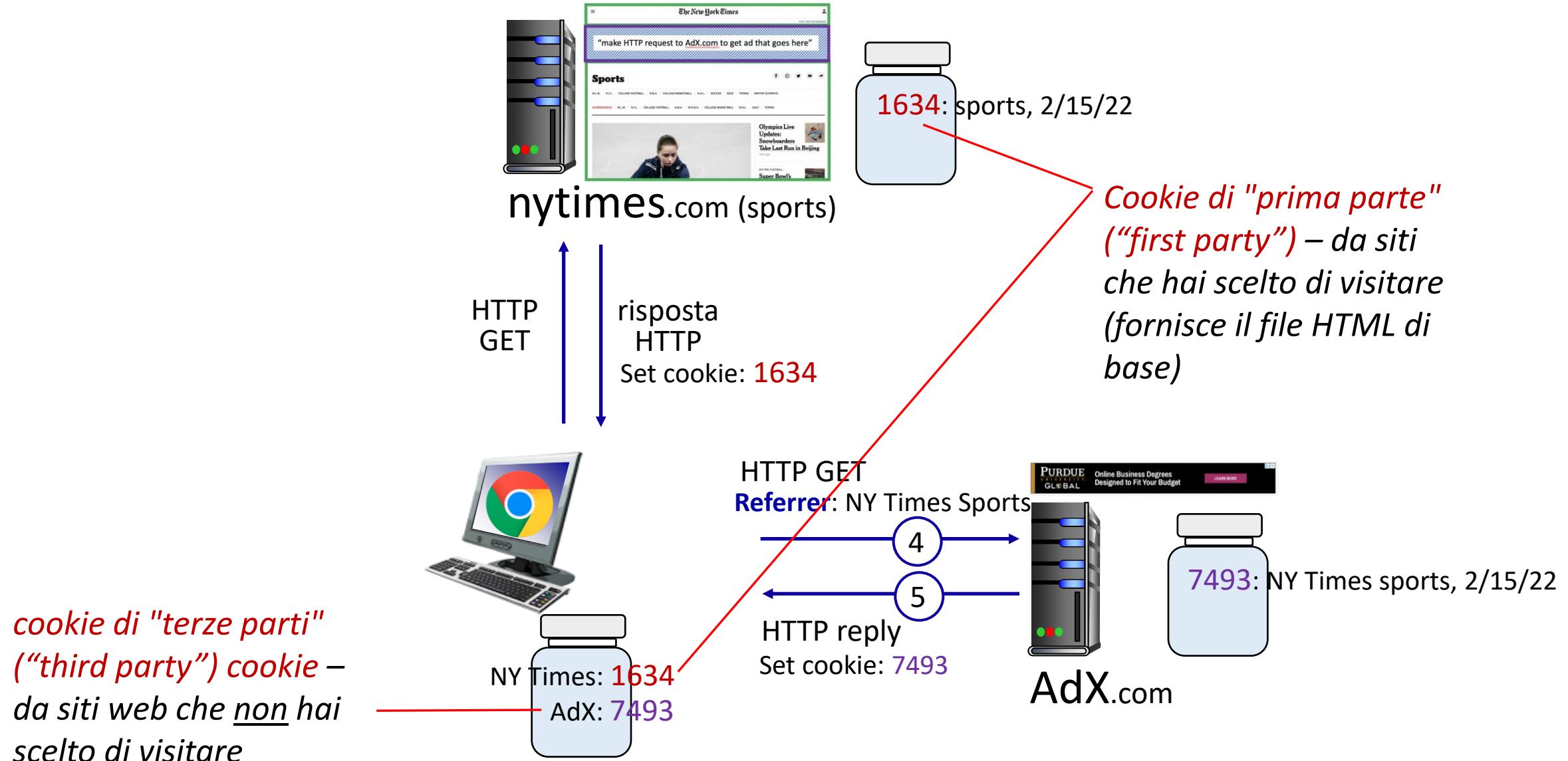
Fonte immagine: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies>

Esempio: visualizzare una pagina web del NY Times

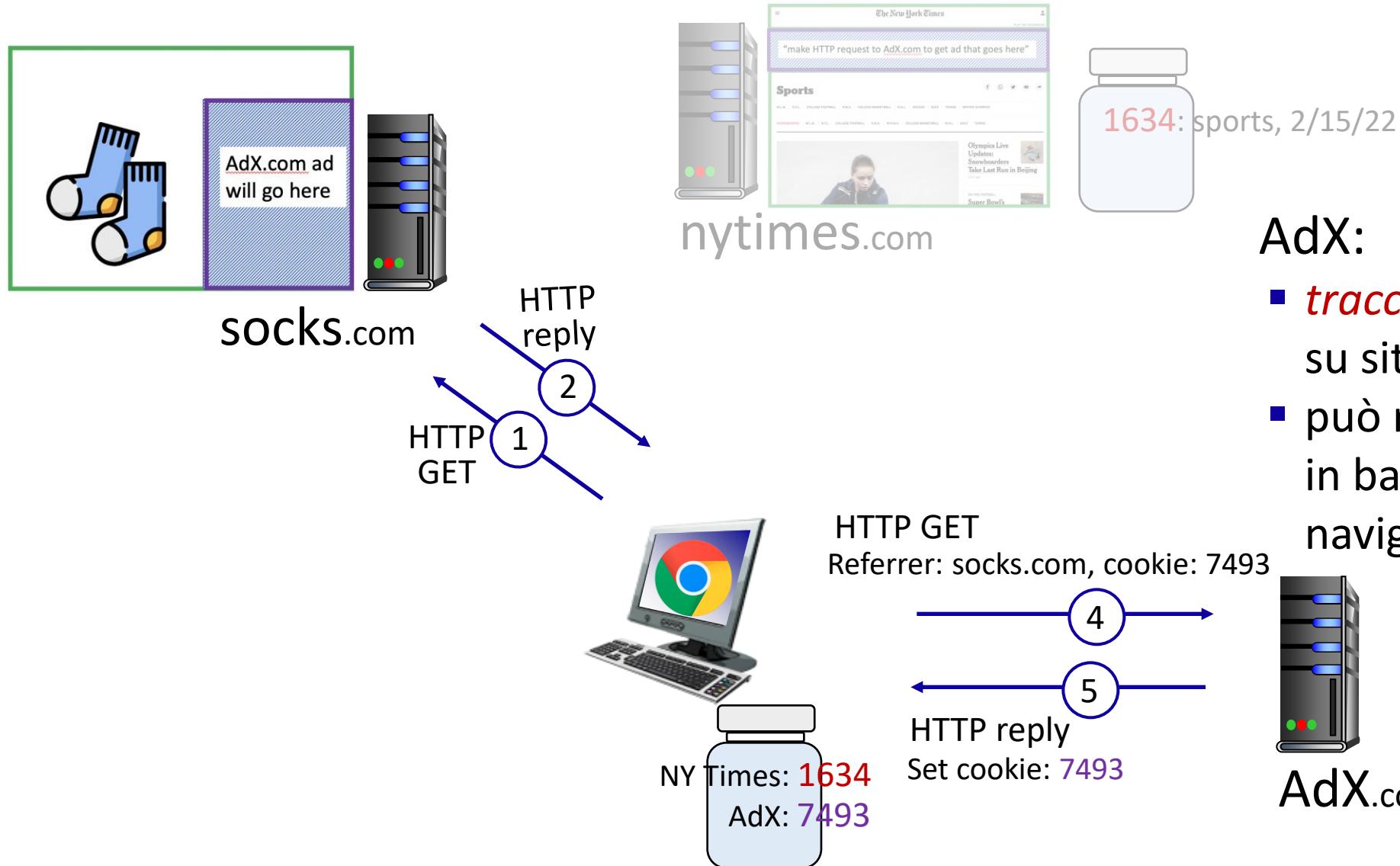
- 1 invia una GET per il file HTML di base da nytimes.com
- 2
- 4 recupera l'annuncio da AdX.com
- 5
- 7 mostra la pagina composta



Cookie: tracciare il comportamento di navigazione di un utente

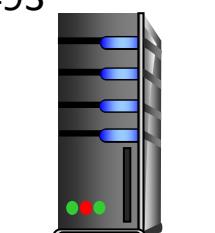


Cookie: tracciare il comportamento di navigazione di un utente



AdX:

- *traccia la mia navigazione* su siti con annunci AdX
- può restituire annunci mirati in base alla cronologia di navigazione



7493: NY Times sports, 2/15/22
7493: socks.com, 2/16/22

AdX.com

Cookie: tracciare il comportamento di navigazione di un utente (un giorno dopo)



HTTP
GET
cookie: **1634**

HTTP
reply
Set cookie: **1634**

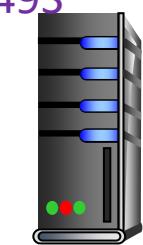


HTTP GET
Referrer: nytimes.com, cookie: **7493**

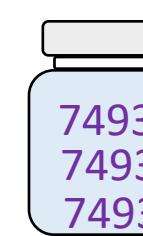
4

5

HTTP reply
Set cookie: **7493**
*Restituito annuncio per
I calzini!*



AdX.com



7493: NY Times sports, 2/15/22
7493: socks.com, 2/16/22
7493: NY Times arts, 2/17/22

Cookie: tracciare il comportamento di navigazione di un utente

I cookie posso essere usati per:

- tracciare (*track*) il comportamento degli utenti su un dato sito (**cookie di prima parte**)
- tracciare il comportamento degli utenti su più siti (**cookie di terze parti**) senza neppure che l'utente abbia mai scelto di visitare il sito del tracker (!)
- il tracciamento può essere *invisibile* all'utente:
 - piuttosto che un annuncio visualizzato che attiva HTTP GET al tracker, potrebbe essere un collegamento invisibile

tracciamento di terze parti tramite cookie:

- disabilitato per impostazione predefinita nei browser Firefox e Safari
- eliminazione graduale dei cookie di terze parti nel browser Chrome, inizialmente bloccati per l'1% degli utenti a partire da Gennaio 2024, con l'obiettivo di estendere il blocco a tutti nel terzo trimestre del 2024

GDPR (EU General Data Protection Regulation)

Regulation - 2016/679 - EN - gd X +

https://eur-lex.europa.eu/eli/reg/2016/679/oj

▼ Expand all ▲ Collapse all

Text

Document information

Procedure

Document summary

Save to My items
 Up-to-date link
 Permanent link
 Download notice
 Follow this document

Table of contents

Hide consolidated versions

04/05/2016 Legal act

Languages, formats and link to OJ

BG ES CS DA DE ET EL EN FR GA HR IT LV LT HU MT NL PL PT RO SK SL FI SV

HTML 

PDF 

Official Journal 

Multilingual display

English (en) Please choose Please choose **Display**

Text

4.5.2016 EN Official Journal of the European Union L 119/1

REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL

of 27 April 2016

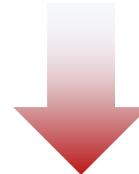
on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)

GDPR (EU General Data Protection Regulation) e i cookie

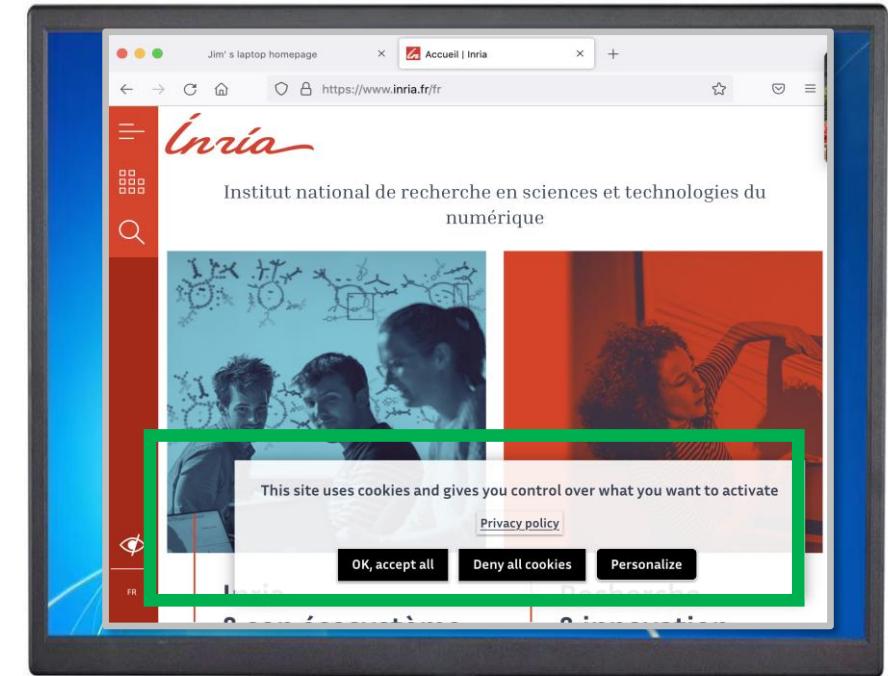
“Le persone fisiche possono essere associate a identificativi online [...], quali gli indirizzi IP, a marcatori temporanei (cookies) o a identificativi di altro tipo, [...].

Tali identificativi possono lasciare tracce che, in particolare se combinate con identificativi univoci e altre informazioni ricevute dai server, possono essere utilizzate per creare profili delle persone fisiche e identificarle.”

GDPR, recital 30 (May 2018)



quando i cookie possono identificare un individuo, i cookie sono considerati dati personali, soggetti alla normativa GDPR sui dati personali

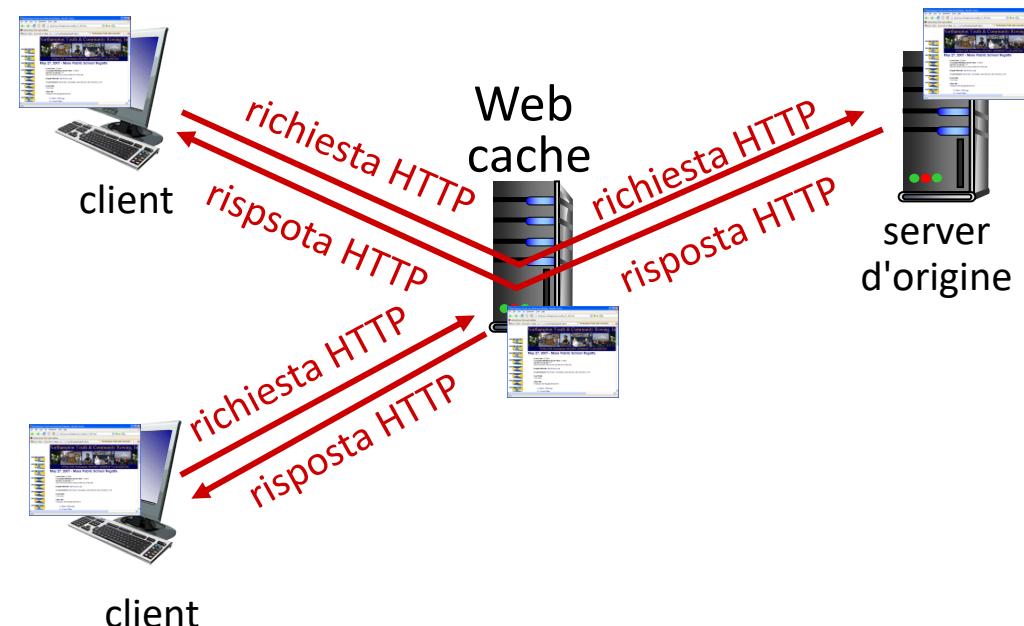


L'utente ha un controllo esplicito sull'autorizzazione o meno dei cookie.

Web cache

Obiettivo: soddisfare la richiesta del client senza coinvolgere il server d'origine (*origin server*)

- l'utente configura il browser per usare una *Web cache* (locale)
- Il browser trasmette tutte le richieste HTTP alla cache
 - *se* l'oggetto è nella cache: la cache fornisce l'oggetto al client
 - *altrimenti* la cache richiede l'oggetto al server d'origine, memorizza ("cache") l'oggetto ricevuto, e infine lo restituisce al client



Web cache (server proxy)

- la cache opera come client (per il server d'origine) e come server (per il client originale)
- Il server comunica alla cache la cache consentita dell'oggetto nell'intestazione della risposta:

```
Cache-Control: max-age=<seconds>
```

```
Cache-Control: no-cache
```

Queste righe di intestazione possono essere usate anche nelle richieste con un significato analogo.

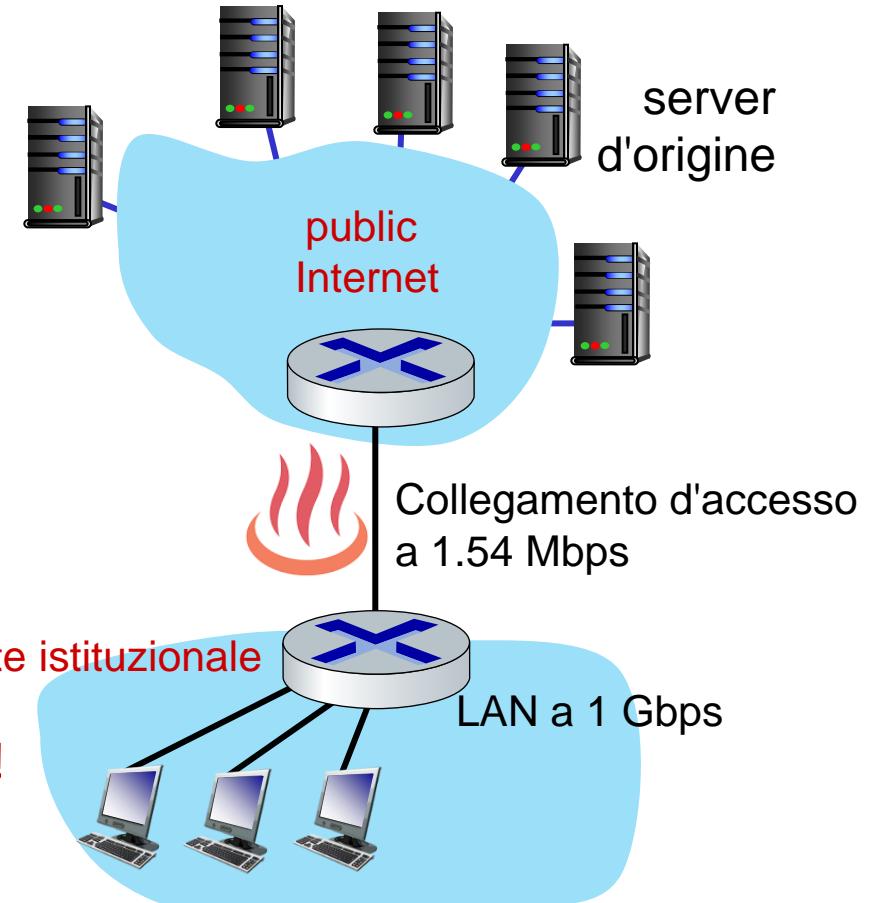
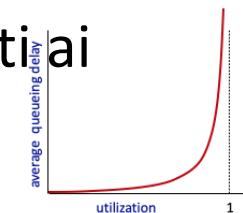
Perché il web caching?

- riduce i tempi di risposta alle richieste dei client
 - la cache è più vicina ai client
- riduce il traffico sul collegamento di accesso a Internet istituzionale
- Riduce il traffico globale su Internet
 - beneficia tutte le applicazioni

Esempio di caching

Scenario:

- velocità collegamento d'accesso: 1.54 Mbps
- RTT da router istituzionale al server: 2 s
- dimensione di un oggetto: 100 kbit
- frequenza media di richieste dai browser istituzionali al server d'origine: 15/s
 - velocità media di trasmissione dei dati ai browser: 1.50 Mbps



Prestazioni:

- utilizzazione del collegamento d'accesso = .97 *problema: ritardo d'accodamento elevato con elevata utilizzazione!*
- utilizzazione della LAN: .0015
- end-end delay = ritardo di Internet + ritardo del collegamento d'accesso + ritardo della LAN
= 2 s + **minuti + microsecondi**

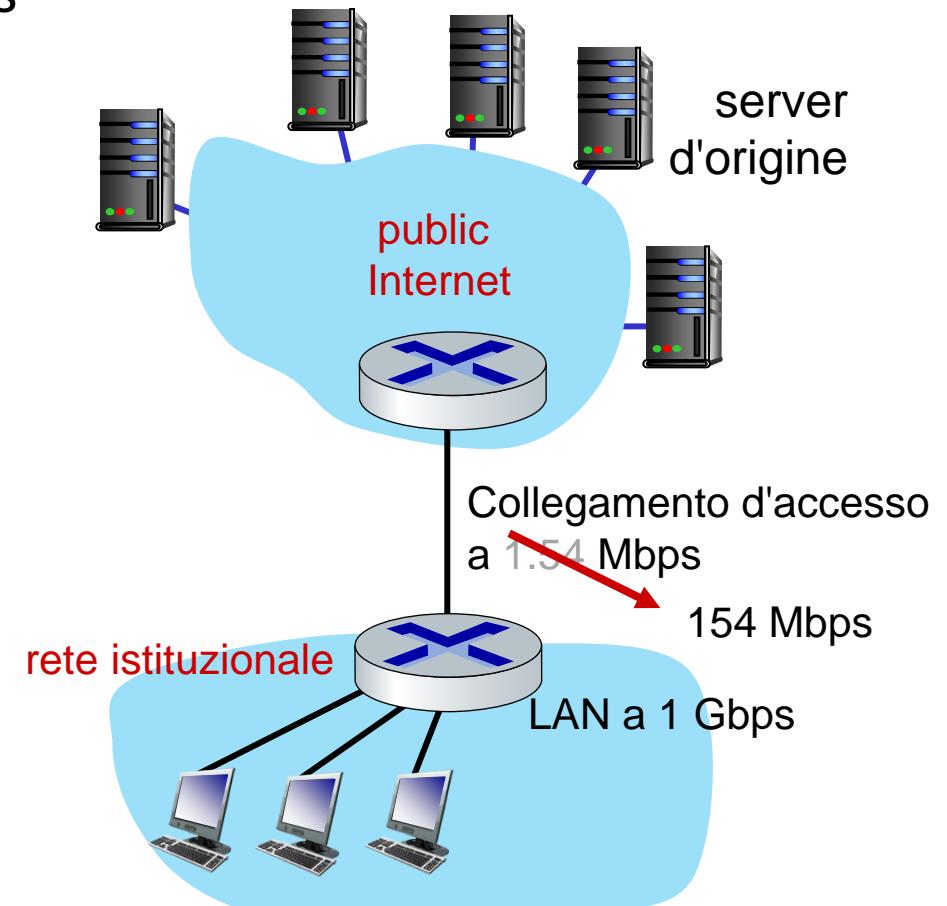
Opzione 1: collegamento d'accesso più veloce

Scenario:

- velocità collegamento d'accesso: ~~1.54~~ Mbps
- RTT da router istituzionale al server: 2 s
- dimensione di un oggetto: 100 kbit
- frequenza media di richieste dai browser istituzionali al server d'origine: 15/s
 - velocità media di trasmissione dei dati ai browser: 1.50 Mbps

Prestazioni:

- utilizzazione del collegamento d'accesso = ~~.97~~ → .0097
- utilizzazione della LAN: .0015
- end-end delay = ritardo di Internet + millisecondi
 - ritardo del collegamento d'accesso + ritardo della LAN
 - = 2 s + ~~minuti~~ + microsecondi



Costo: collegamento d'accesso più veloce (costoso!)

Opzione 2: installare un web cache

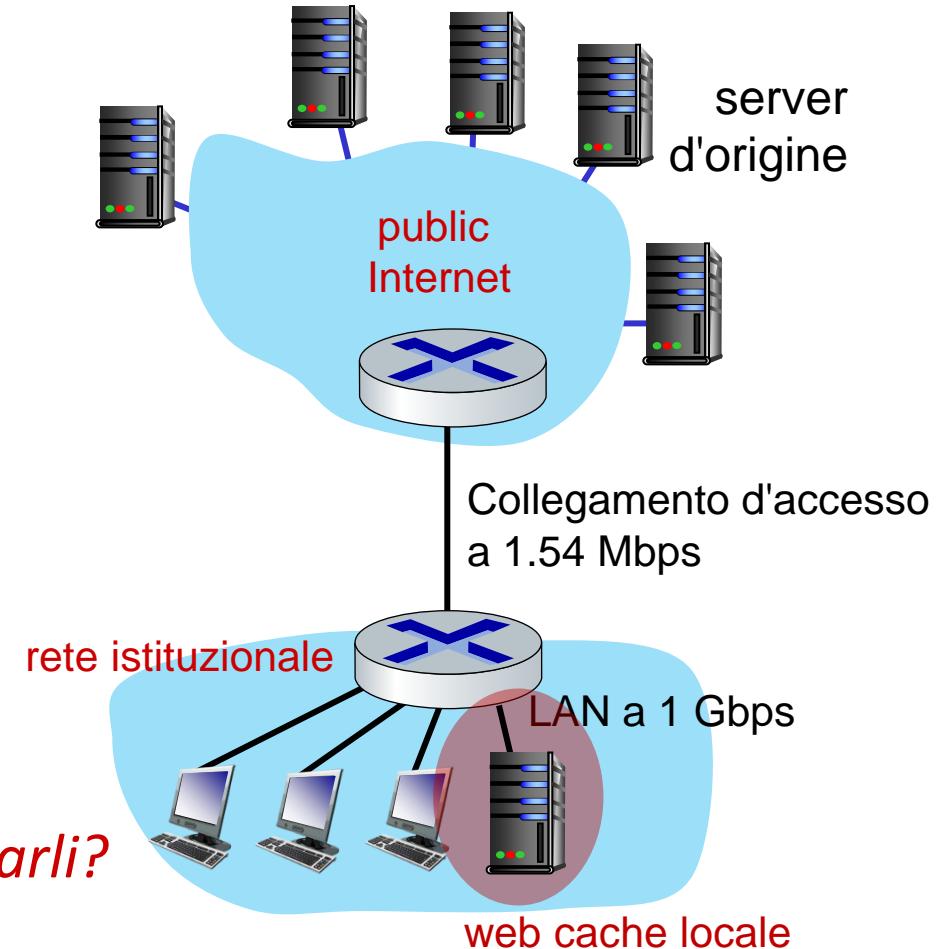
Scenario:

- velocità collegamento d'accesso: 1.54 Mbps
- RTT da router istituzionale al server: 2 s
- dimensione di un oggetto: 100 kbit
- frequenza media di richieste dai browser istituzionali al server d'origine: 15/s
 - velocità media di trasmissione dei dati ai browser: 1.50 Mbps

Costo: web cache (economica!)

Prestazioni:

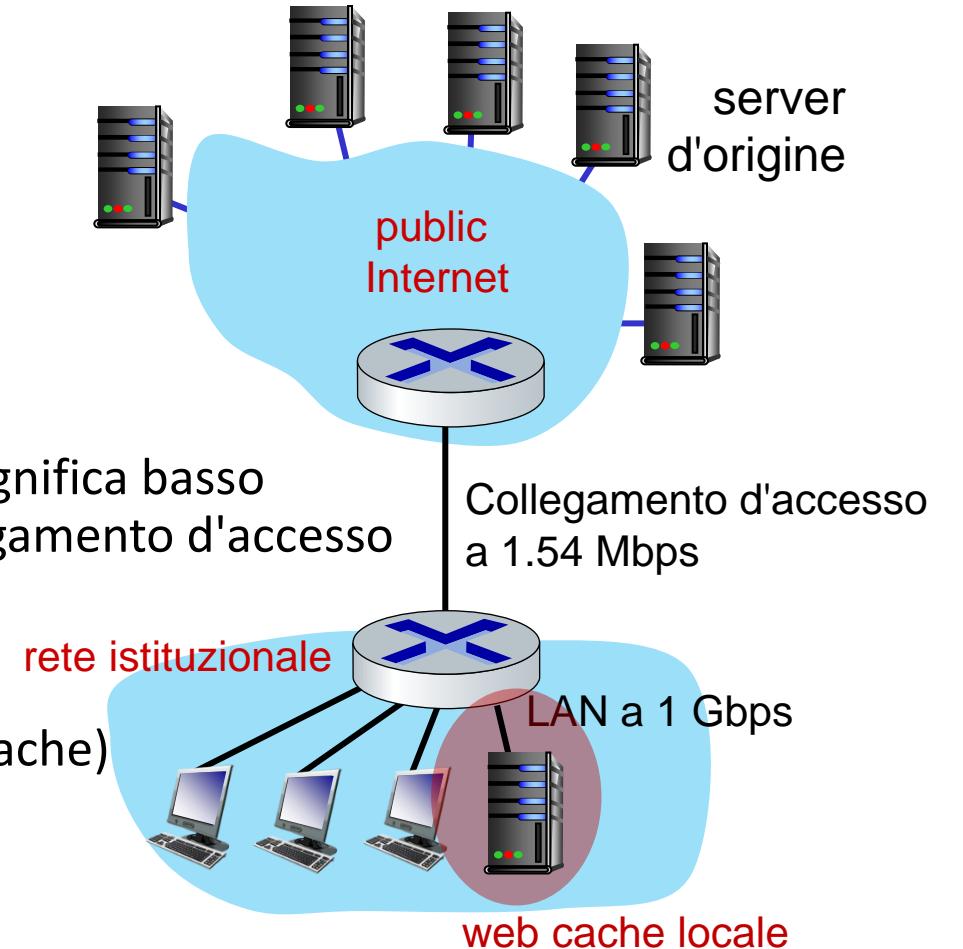
- utilizzazione LAN: .?
- utilizzazione del link di accesso = ? *come calcolarli?*
- ritardo end-end medio = ?



Calcolo dell'utilizzo del collegamento di accesso e del ritardo end-end con la cache:

supponiamo una percentuale di successo (hit rate) pari a 0.4:

- il 40% delle richieste sarà soddisfatto dalla cache, con ritardo basso (millisecondi)
- 60% delle richieste sarà soddisfatto dal server d'origine
 - tasso di trasmissione sul collegamento d'accesso
$$= 0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$$
 - utilizzazione collegamento d'accesso $= 0.9/1.54 = .58$ significa basso (decine di millisecondi) ritardo di accodamento al collegamento d'accesso
- ritardo end-end medio:
$$\begin{aligned} &= 0.6 * (\text{ritardo dai server d'origine}) \\ &\quad + 0.4 * (\text{ritardo quando richiesta soddisfatta dalla cache}) \\ &= 0.6 (2.01) + 0.4 (\sim\text{ms}) = \sim 1.2 \text{ s} \end{aligned}$$

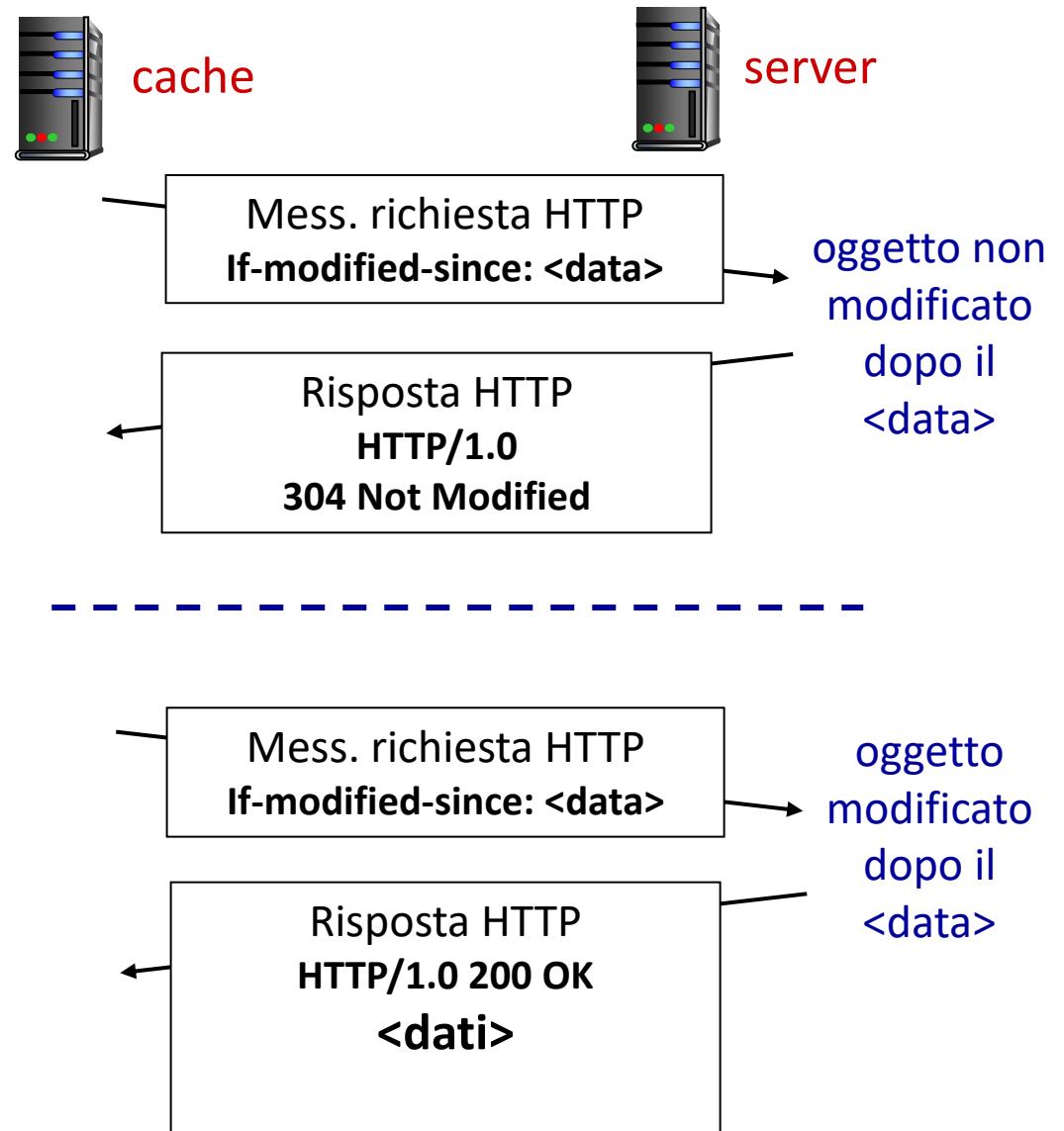


ritardo medio end-end inferiore che con un collegamento a 154 Mbps (e meno costoso!)

GET condizionale

Obiettivo: non inviare un oggetto se la cache ha una copia aggiornata dell'oggetto

- Nessun ritardo di trasmissione dell'oggetto (o uso delle risorse di rete)
- **client:** specifica la data della copia dell'oggetto nella richiesta HTTP
If-modified-since: <data>
- **server:** la risposta non contiene l'oggetto se la copia nella cache è aggiornata:
HTTP/1.0 304 Not Modified



Nota sul caching

Il caching può essere effettuato da:

- una web cache, ossia uno speciale tipo di proxy, cui il browser invia le richieste invece che indirizzarle all'origin server.
- oppure, dal browser stesso, che conserva una copia degli oggetti richiesti in precedenza

In entrambi i casi, occorre prestare attenzione al problema dell'aggiornamento degli oggetti: vedi riga di intestazione *Cache-Control* e *GET condizionale*.

HTTP/2

Obiettivo principale: diminuzione del ritardo nelle richieste HTTP a più oggetti

HTTP1.1: ha introdotto GET multiple in pipeline su una singola connessione TCP

- il server risponde *in ordine* (FCFS: first-come-first-served scheduling) alle richieste GET
- con FCFS, oggetti piccoli possono dover aspettare per la trasmissione (**head-of-line (HOL) blocking** [blocco in testa alla coda]) dietro a uno o più oggetti grandi
- il recupero delle perdite (ritrasmissione dei segmenti TCP persi) blocca la trasmissione degli oggetti

HTTP/2

Obiettivo principale: diminuzione del ritardo nelle richieste HTTP a più oggetti

HTTP/2: [RFC 7540, 2015] maggiore flessibilità del server nell'invio di oggetti al client:

- metodi, codice di stato, maggior parte dei campi di intestazione inalterati rispetto a HTTP 1.1
- ordine di trasmissione degli oggetti richiesti basata su una priorità degli oggetti specificata dal client (non necessariamente FCFS)
- invio *push* al client di oggetti aggiuntivi, senza che il client li abbia richiesti
- dividere gli oggetti in frame, intervallare i frame per mitigare il blocco HOL

HTTP/2

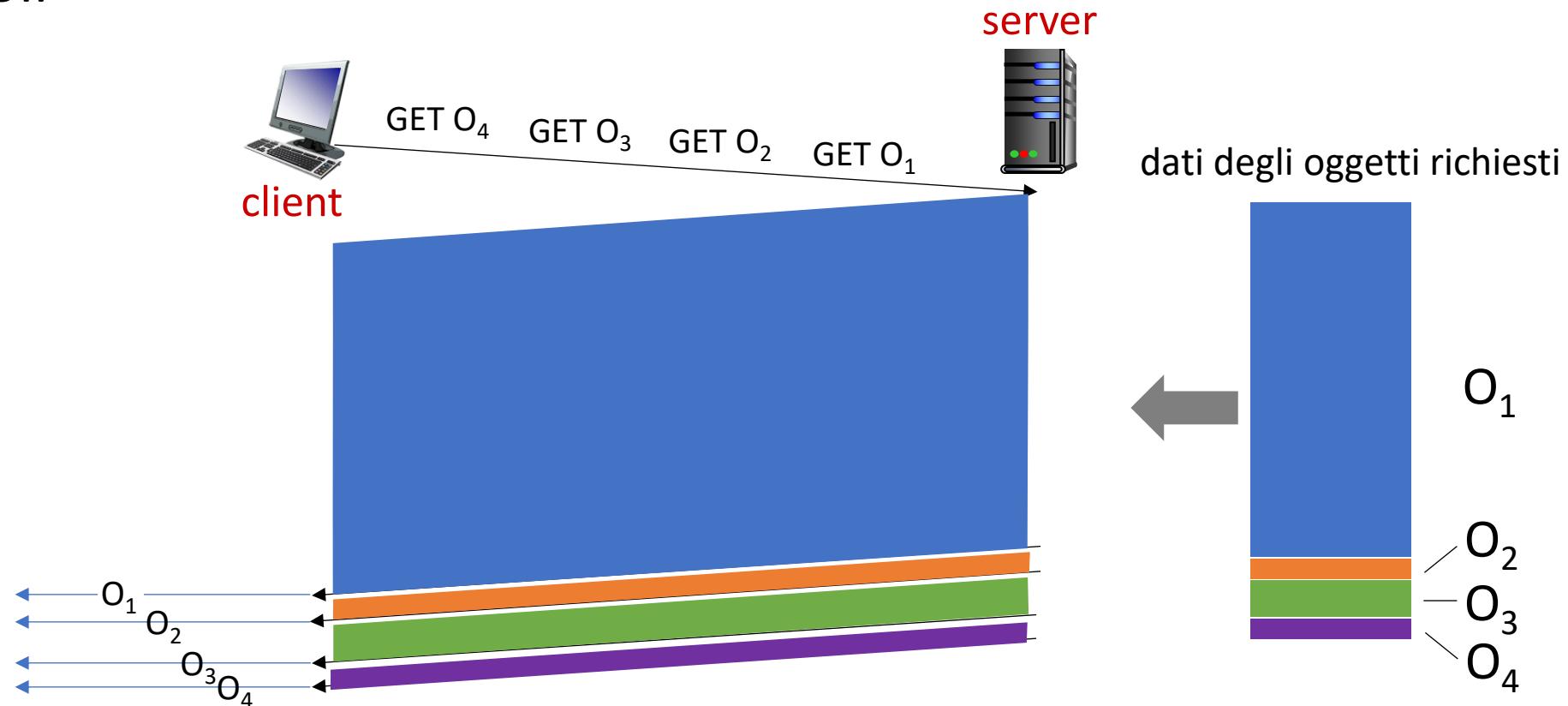
Obiettivo principale: diminuzione del ritardo nelle richieste HTTP a più oggetti

Affrontando l'HOL, HTTP/2 vuole permettere di scaricare una pagina con più oggetti attraverso una singola connessione TCP:

- minor overhead sul server
- miglior funzionamento del controllo della congestione TCP

HTTP/2: mitigazione del blocco HOL

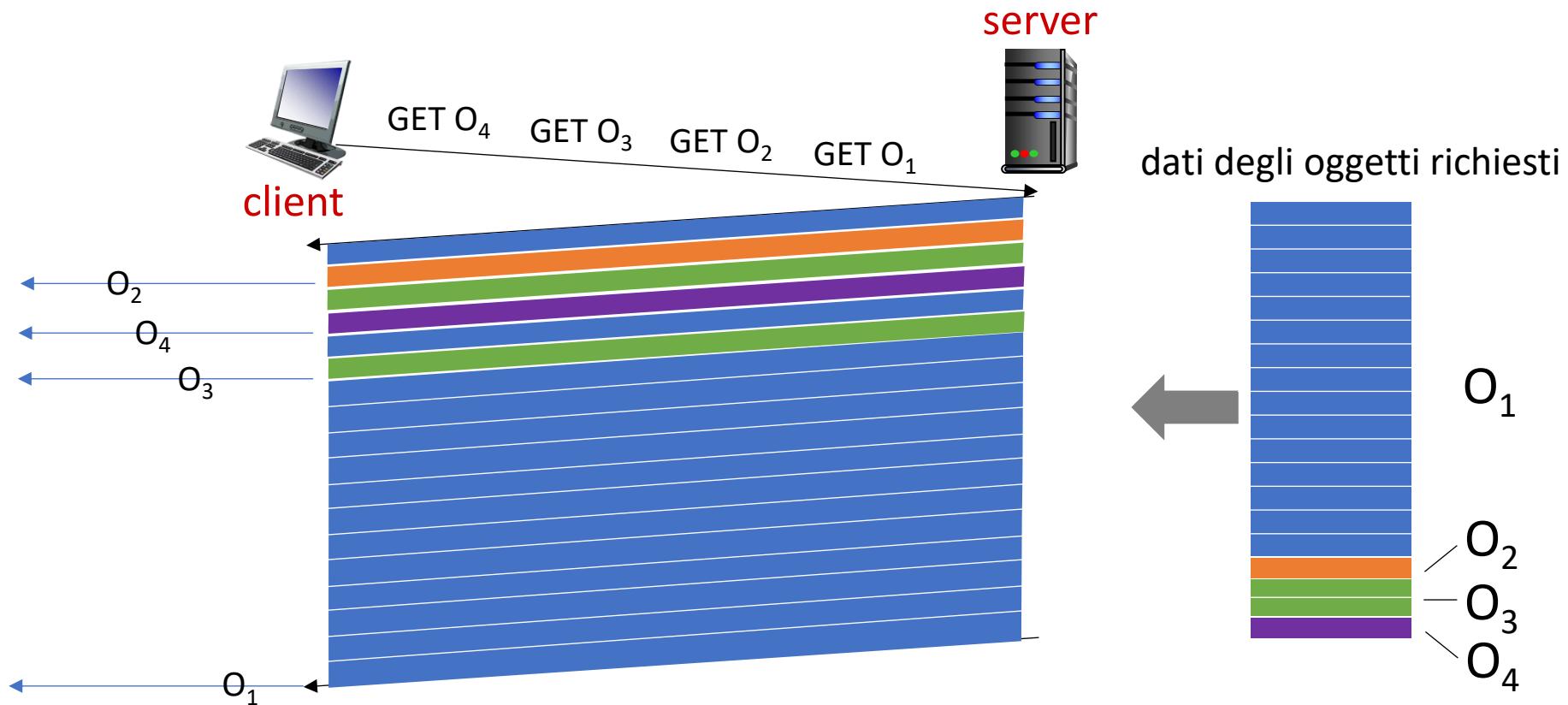
HTTP 1.1: il client richiede 1 oggetto grande (es., file video) e 3 oggetti più piccoli



oggetti consegnati nell'ordine in cui sono stati richiesti: O_2 , O_3 , O_4 aspettano dietro O_1

mitigazione

HTTP/2: oggetti divisi in frame, trasmissione de frame interlacciata



O_2, O_3, O_4 consegnati rapidamente, O_1 leggermente ritardato

Da HTTP/2 a HTTP/3

HTTP/2 su una singola connessione TCP significa:

- il recupero dalla perdita di pacchetti blocca comunque tutte le trasmissioni di oggetti
 - come in HTTP 1.1, i browser sono incentivati ad aprire più connessioni TCP parallele per ridurre lo stallo e aumentare il throughput complessivo
- nessuna sicurezza su una connessione TCP semplice
- **HTTP/3**: aggiunge sicurezza, controllo di errore e congestione per oggetto (più pipelining) su UDP
 - ulteriori informazioni su HTTP/3 trattando il livello di trasporto

Università degli Studi di Roma "Tor Vergata"
Laurea in Informatica

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2024/2025

Livello di applicazione **(parte2)**

dr. Manuel Fiorelli

manuel.fiorelli@uniroma2.it

<https://art.uniroma2.it/fiorelli>

Basate sulle slide del libro di testo:

https://gaia.cs.umass.edu/kurose_ross/ppt.php

Introduction: 1-62

Livello di applicazione: panoramica

- Principi delle applicazioni di rete
- **Web e HTTP**
- E-mail, SMTP, IMAP
- DNS: il servizio di directory di Internet
- Applicazioni P2P
- Streaming video e reti di distribuzione di contenti
- Programmazione delle socket programming con UDP e TCP



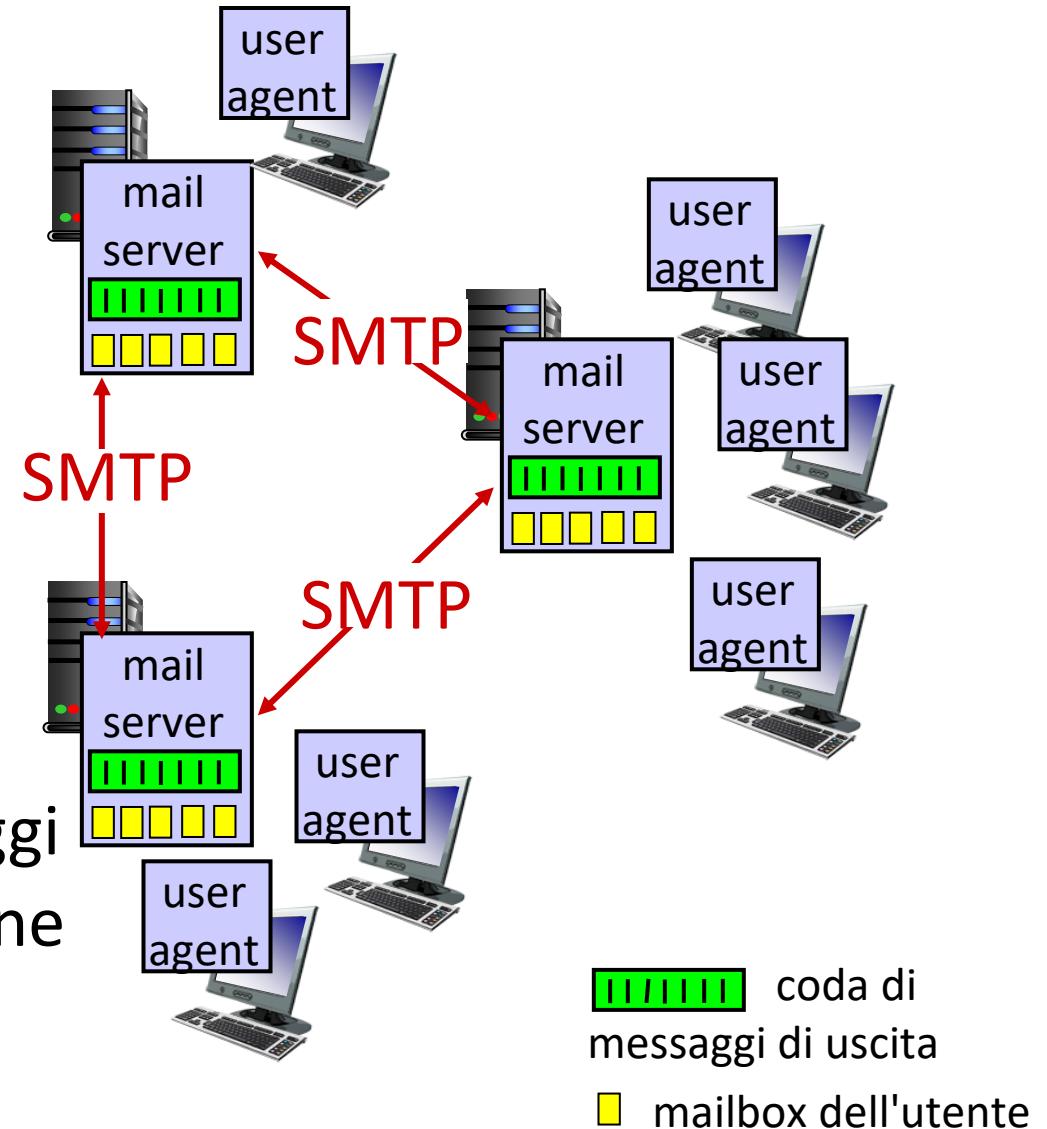
E-mail

Tre componenti principali:

- user agents (o *agenti utenti*)
- mail servers (o *server di posta*)
- simple mail transfer protocol: SMTP

User Agent

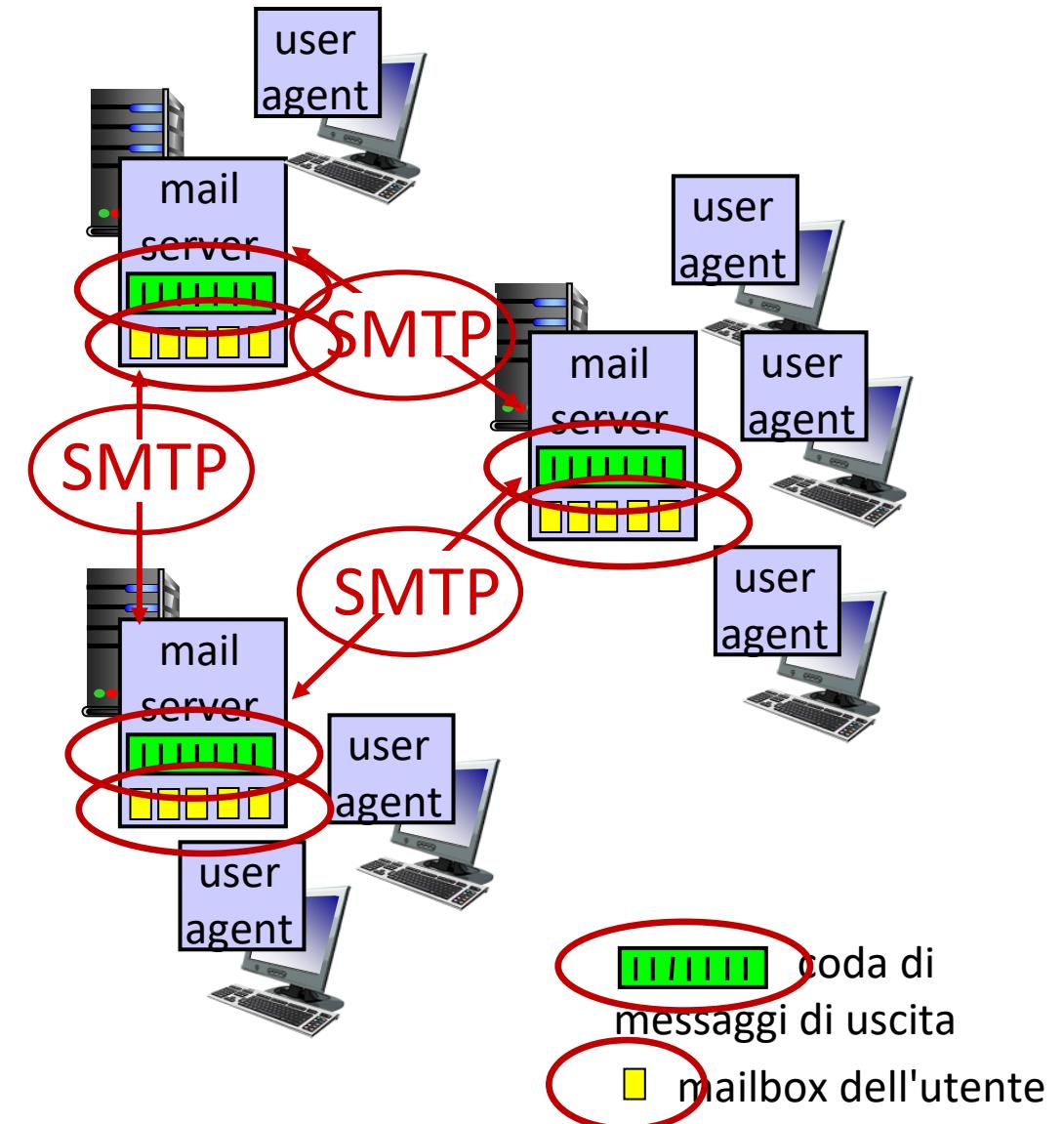
- detto anche “mail reader”
- composizione, editing, lettura dei messaggi
- esempi: Outlook, client di posta dell'iPhone
- i messaggi in uscita o in arrivo sono memorizzati sul server



E-mail: mail servers

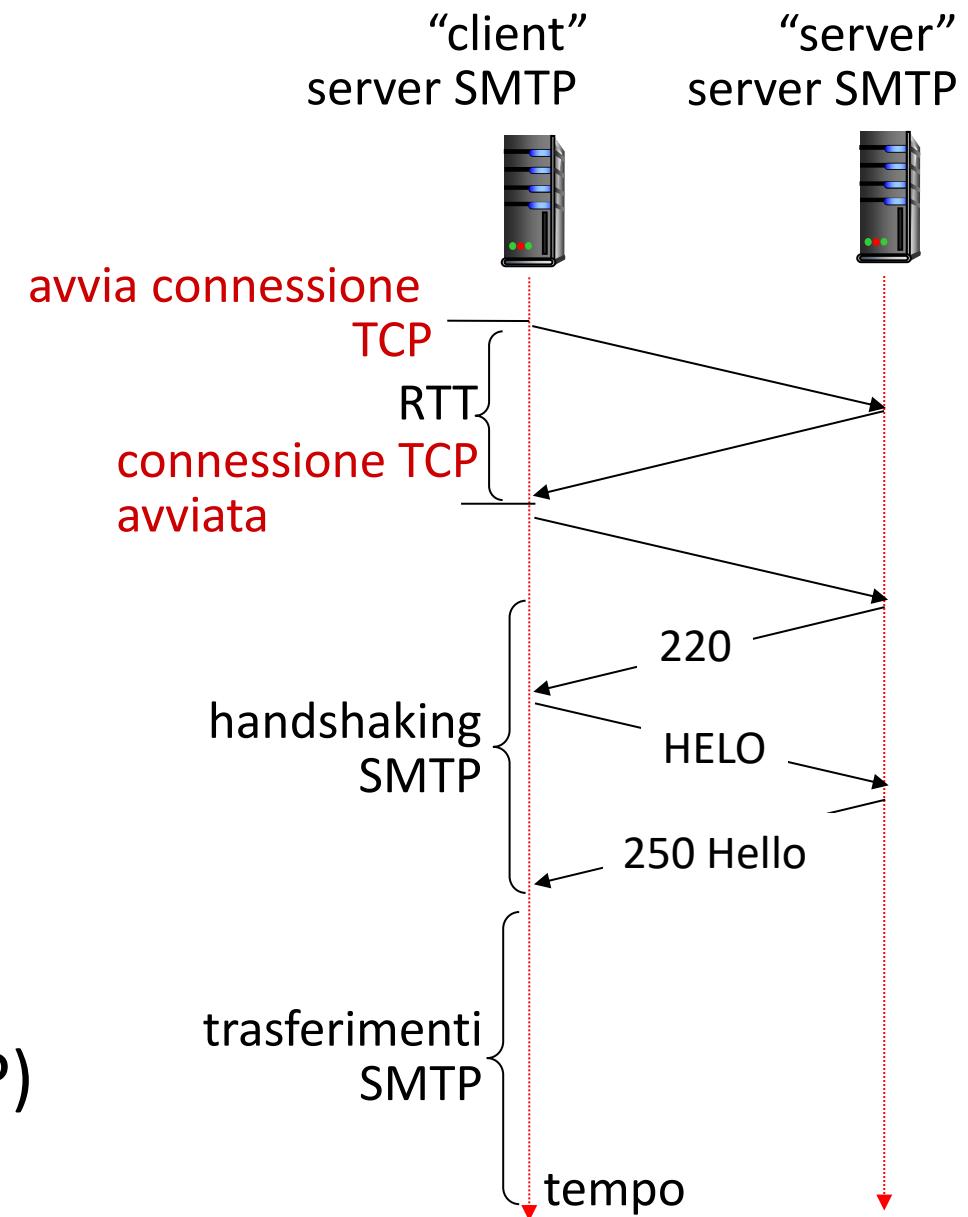
mail server:

- *mailbox* (casella di posta) contiene i messaggi in arrivo per l'utente
 - *coda di messaggi* da trasmettere
- protocollo SMTP** tra mail server per inviare messaggi email
- client: mail server trasmittente
 - “server”: mail server ricevente



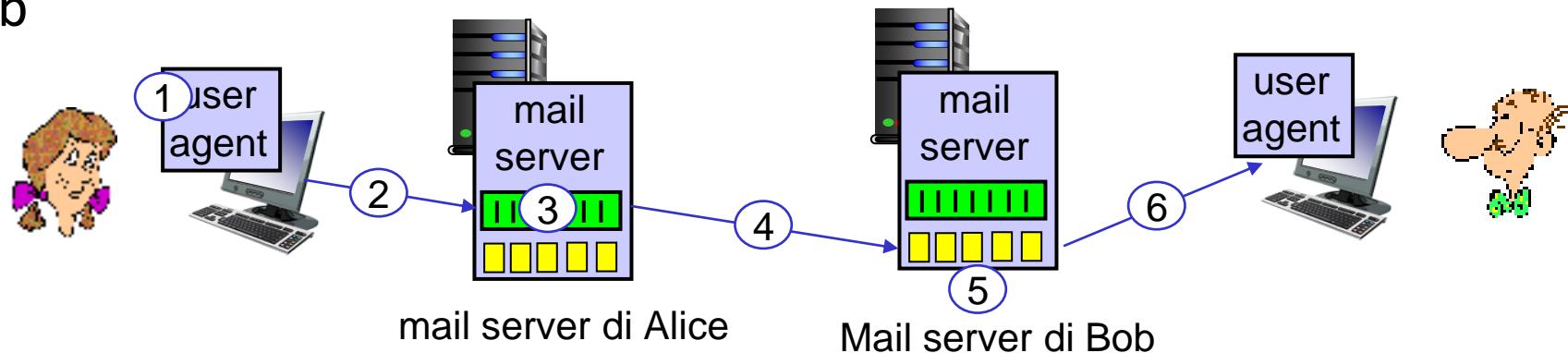
SMTP RFC (5321)

- usa TCP per trasferire un modo affidabile i messaggi di posta elettronica dal client (mail server che avvia la connessione) al server, porta 25
 - Trasferimento diretto: il server trasmittente al server ricevente
- Tre fasi per il trasferimento
 - handshaking (saluto)
 - trasferimento dei messaggi
 - chiusura
- Interazione comando/risposta (come HTTP)
 - comandi: testo ASCII a 7 bit
 - risposta: codice di stato e espressione



Scenario: Alice invia un'e-mail a Bob

- 1) Alice usa il suo user agent per comporre il messaggio da inviare "a" ("to") bob@someschool.edu
- 2) lo user agent di Alice invia un messaggio al server di posta di Alice; il messaggio è posto nella coda di messaggi
- 3) il lato client di SMTP apre una connessione TCP con il mail server di Bob



- 4) il client SMTP invia il messaggio di Alice sulla connessione TCP
- 5) il mail server di Bob pone il messaggio nella casella di posta di Bob
- 6) Bob invoca il suo user agent per leggere il messaggio

Esempio di interazione SMTP

S: 220 hamburger.edu

SMTP: note finali

confronto con HTTP:

- HTTP: client pull
- SMTP: client push
- Entrambi hanno un'interazione comando/risposta in ASCII, codici di stato
- HTTP: ciascun oggetto è encapsulato nel suo messaggi di risposta
- SMTP: più oggetti vengono trasmessi in un unico messaggio

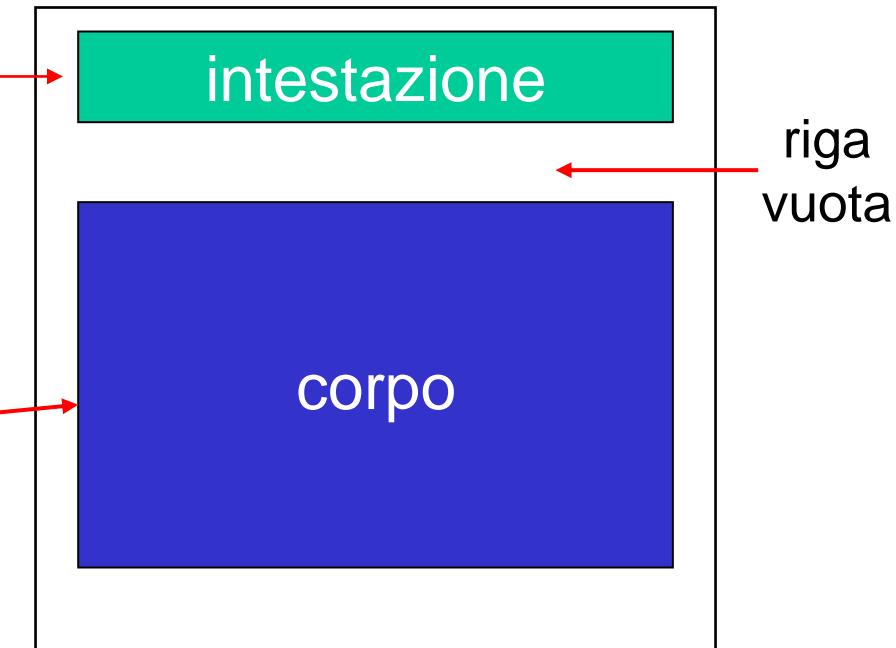
- SMTP usa connessione persistenti
- SMTP richiede che il messaggio (intestazione e corpo) sia nel formato ASCII a 7 bit
- Il server SMTP usa CRLF.CRLF per determinare la fine del messaggio

Formato dei messaggi di posta elettronica

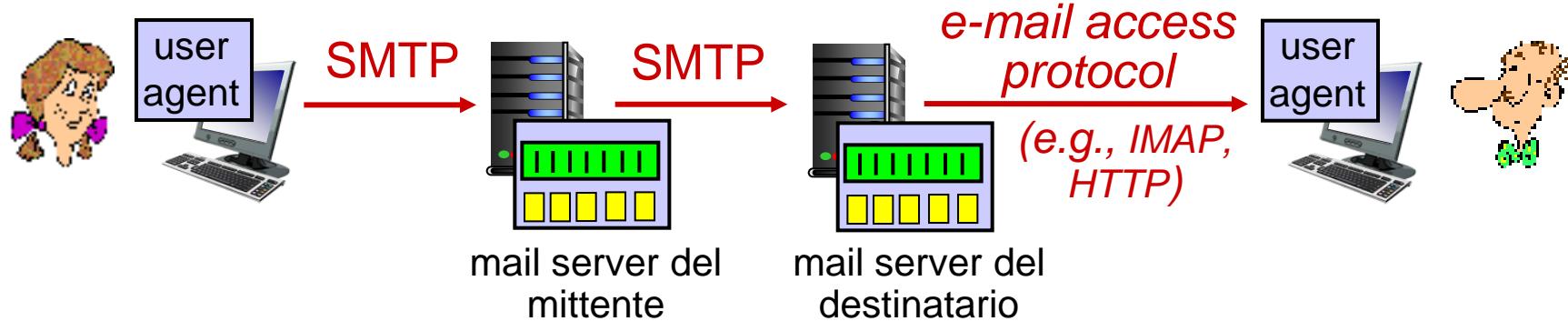
SMTP: protocollo per scambiare messaggi di posta elettronica, definito nell'RFC 5321 (come RFC 7231 definisce HTTP)

RFC 2822 definisce la *sintassi* dei messaggi di posta elettronica (come HTML definisce la sintassi per i documenti web)

- Righe di intestazione, per esempio.,
 - To/A:
 - From/Da:
 - Subject/Oggetto:
differenti da comandi SMTP MAIL FROM:, RCPT TO:!
- corpo: il “messaggio”, soltanto caratteri ASCII



Protocolli di accesso alla posta



- **SMTP:** consegna/memorizzazione sul server del destinatario
- protocollo di accesso alla posta: ottenere i messaggi dal server
 - **IMAP:** Internet Mail Access Protocol [RFC 3501]: messaggi memorizzati sul server, IMAP consente di recuperare, cancellare e archiviare i messaggi memorizzati sul server.
- **HTTP:** gmail, Hotmail, Yahoo!Mail, etc. consente interfaccia web sopra a STMP (per l'invio) e IMAP (o POP) per il recupero delle email.

Università degli Studi di Roma "Tor Vergata"
Laurea in Informatica

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2023/2024

Livello di applicazione (parte3)

dr. Manuel Fiorelli

manuel.fiorelli@uniroma2.it

<https://art.uniroma2.it/fiorelli>

Basate sulle slide del libro di testo:

https://gaia.cs.umass.edu/kurose_ross/ppt.php

Introduction: 1-72

Livello di applicazione: panoramica

- Principi delle applicazioni di rete
- Web e HTTP
- E-mail, SMTP, IMAP
- DNS: il servizio di directory di Internet
- Applicazioni P2P
- Streaming video e reti di distribuzione di contenti
- Programmazione delle socket programming con UDP e TCP



Problema: risoluzione dei nomi

persone: molti identificatori:

- Nome, codice fiscal,
numero della carta di
identità

Host e router di Internet:

- indirizzo IP (32 bit) - usato per
indirizzare i datagrammi
- “nome”, ad esempio,
cs.umass.edu - usato dagli esseri
umani

D: come mappare tra indirizzo
IP e nome e viceversa?

Problema: risoluzione dei nomi

persone: molti identificatori:

- Nome, codice fiscal, numero della carta di identità

Host e router di Internet:

- indirizzo IP (32 bit) - usato per indirizzare i datagrammi
- “nome”, ad esempio, cs.umass.edu - usato dagli esseri umani

D: come mappare tra indirizzo IP e nome e viceversa?

File hosts (`/etc/hosts` nei sistemi POSIX)

Associa un indirizzo IP a uno o più hostname

```
185.300.10.1 host1
185.300.10.2 host2 merlin
185.300.10.3 host3 arthur king
185.300.10.4 timeserver
```

Locale a un nodo, il suo contenuto non deve necessariamente coincidere con quello di altri nodi (ma meglio evitarlo!)

Problema: risoluzione dei nomi

persone: molti identificatori:

- Nome, codice fiscal, numero della carta di identità

Host e router di Internet:

- indirizzo IP (32 bit) - usato per indirizzare i datagrammi
- “nome”, ad esempio, cs.umass.edu - usato dagli esseri umani

D: come mappare tra indirizzo IP e nome e viceversa?

Anni '70: HOSTS.TXT

- *Mantenuto dal Network Information Center (NIC) presso lo SRI*
- *Reso disponibile su un host designato* (attraverso il protocollo FTP)
- *Installato dagli amministratori di sistema sui singoli nodi*

Problemi:

- *Crescita del file*
- *Traffico generato sull'host dove era pubblicato*

DNS: Domain Name System

persone: molti identificatori:

- nome, codice fiscale, numero della carta di identità

Host e router di Internet:

- indirizzo IP (32 bit) - usato per indirizzare i datagrammi
- “nome”, ad esempio, cs.umass.edu - usato dagli esseri umani

D: come mappare tra indirizzo IP e nome e viceversa?

Domain Name System (DNS):

- *Database distribuito* implementato in una gerarchia di *name server*
- *Protocollo a livello di applicazione* che consente agli host, ai router e ai server DNS di comunicare per *risolvere* i nomi (traduzione nome/indirizzo)
 - *si noti*: funzioni critiche di Internet, **implementate come protocollo a livello di applicazione**
 - complessità nelle parti periferiche della rete

DNS: servizi, struttura

Servizi DNS

- Traduzione degli hostname in indirizzi IP
- host aliasing
 - nome canonico e alias
- mail server aliasing
- load distribution (*distribuzione del carico di rete*)
 - server Web replicati: più indirizzi IP corrispondono a un solo nome

Q: Perché non centralizzare il DNS?

- un *single point of failure* (*punto di vulnerabilità*)
- volume di traffico
- database centralizzato distante
- manutenzione

R: non scala!

- Solo i server DNS di Comcast: 600B query DNS al giorno
- Solo i server DNS Akamai: 2,2T query DNS al giorno

Pensare al DNS

un enorme database distribuito:

- ~ miliardi di record, ciascuno semplice

gestisce molti *trilioni* di interrogazioni al giorno :

- *molte* più letture che scritture
- *è importante*: quasi tutte le transazioni Internet interagiscono con il DNS - i millisecondi contano!

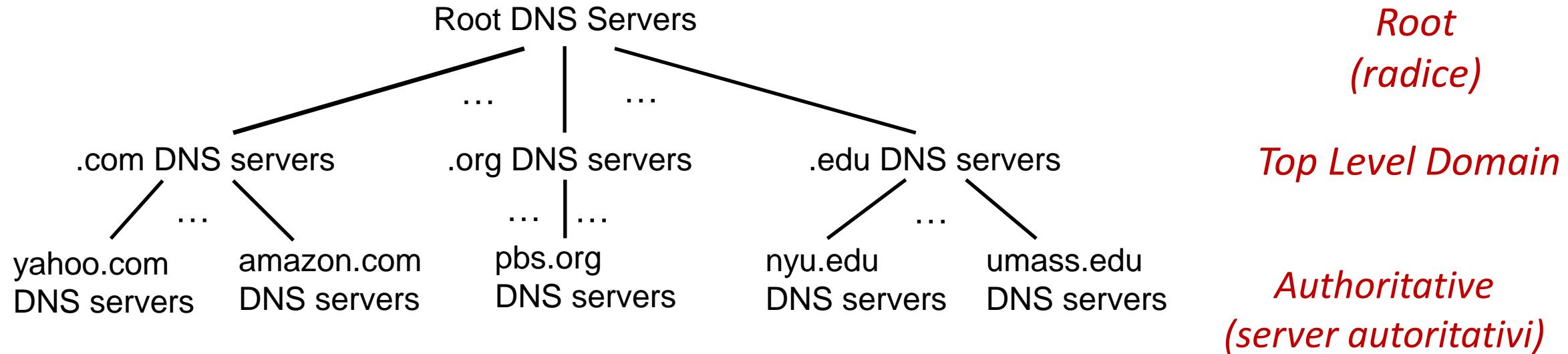
organizzativamente e fisicamente decentralizzato

- milioni di organizzazioni diverse responsabili dei loro *record*

"a prova di proiettile": affidabilità, sicurezza



DNS: un database distribuito e gerarchico

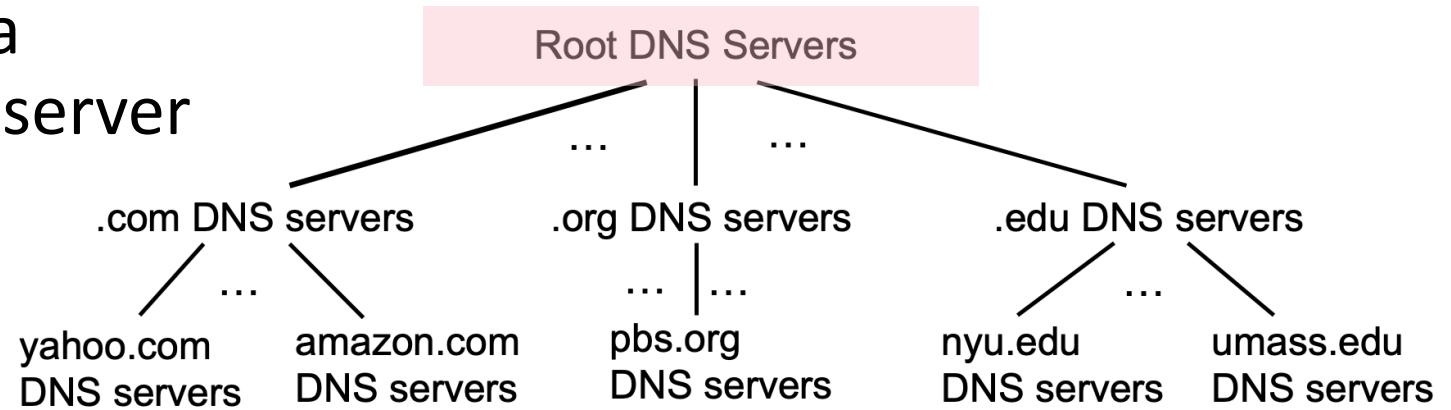


Il client vuole l'indirizzo IP di www.amazon.com; 1^a approssimazione:

- il cliente interroga il root server per trovare il TLD server per .com
- il client interroga il TLD server .com per ottenere il server autoritativo per amazon.com
- il client interroga il server autoritativo per amazon.com per ottenere l'indirizzo IP di www.amazon.com

DNS: root name server

- ufficiale, contatto di ultima istanza da parte dei name server che non sono in grado di risolvere il nome



DNS: root name server

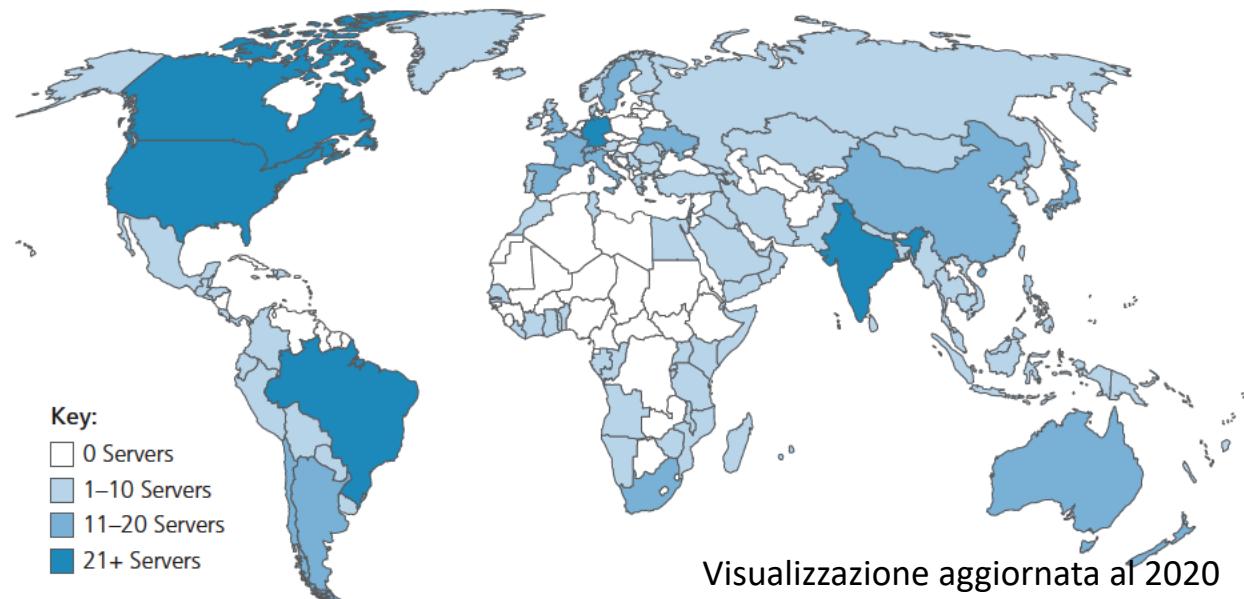
- ufficiale, contatto di ultima istanza da parte dei name server che non sono in grado di risolvere il nome.

Fornisce gli indirizzi IP dei TLD server

- funzione *incredibilmente importante* di Internet
 - Internet non potrebbe funzionare senza!
 - DNSSEC – offre sicurezza (autenticazione, integrità dei messaggi)
- ICANN (Internet Corporation for Assigned Names and Numbers) gestisce il root DNS domain

13 name "server" logici in tutto il mondo, ogni "server" replicato più volte (~200 server negli USA)

<https://www.internic.net/domain/named.root>



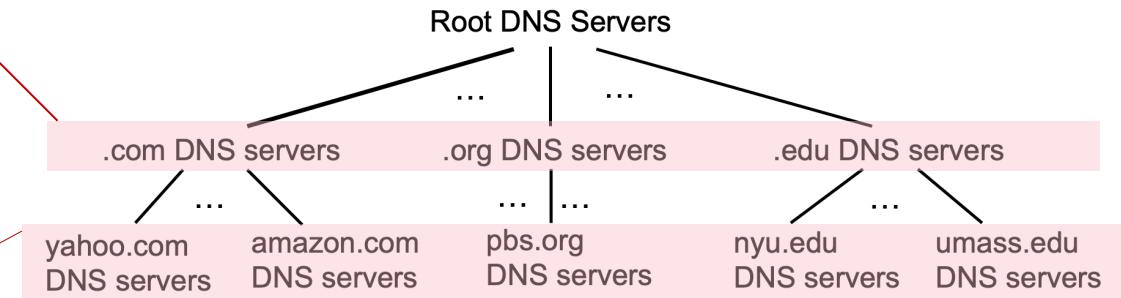
Visualizzazione aggiornata al 2020

Il 20/03/2023 ci sono 1813 istanze gestite da 12 operatori, coordinate dallo IANA (fonte: <https://root-servers.org/>)

Top-Level Domain, and authoritative servers

Top-Level Domain (TLD) DNS server:

- si occupano dei domini .com, .org, .net, .edu, .aero, .jobs, .museums, e di tutti i domani locali di alto livello, quali .cn, .uk, .fr, .ca, .jp
- Network Solutions: gestisce i server TLD per i domini .com e .net
- Educause: gestisce quelli per .edu



DNS server autoritativo:

- server DNS propri di ciascuna organizzazione, che forniscono i mapping ufficiali da hostname a IP per gli host dell'organizzazione
- possono essere mantenuti dall'organizzazione o dal service provider

Name server DNS locali

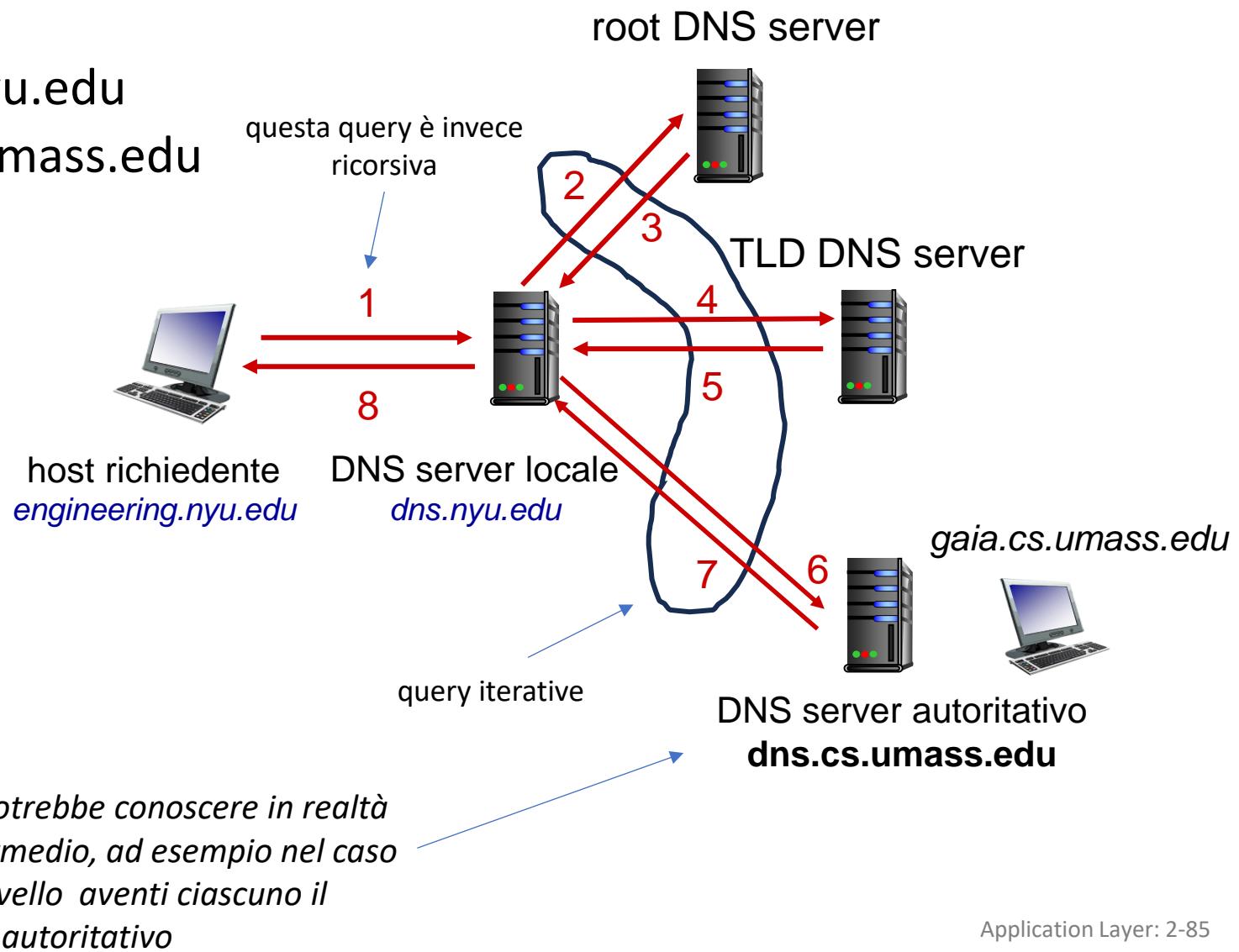
- quando l'host effettua una richiesta DNS, la query viene inviata al suo server DNS *locale* (con funzione di *default name server*)
 - il server DNS locale restituisce una risposta, rispondendo:
 - dalla sua cache locale di coppie nome->indirizzo (possibilmente non aggiornate!)
 - inoltrando la richiesta alla gerarchia DNS per la risoluzione
 - ciascun ISP ha un proprio server DNS locale; per trovare il vostro:
 - MacOS: % scutil --dns
 - Windows: >ipconfig /all
- il server DNS locale non appartiene strettamente alla gerarchia dei server

DNS: interrogazione iterativa

Esempio: l'host engineering.nyu.edu vuole l'indirizzo IP di gaia.cs.umass.edu

Query iterativa

- Il server contattato risponde con il nome del server da contattare
- “Io non conosco questo nome, ma puoi chiederlo a questo server”

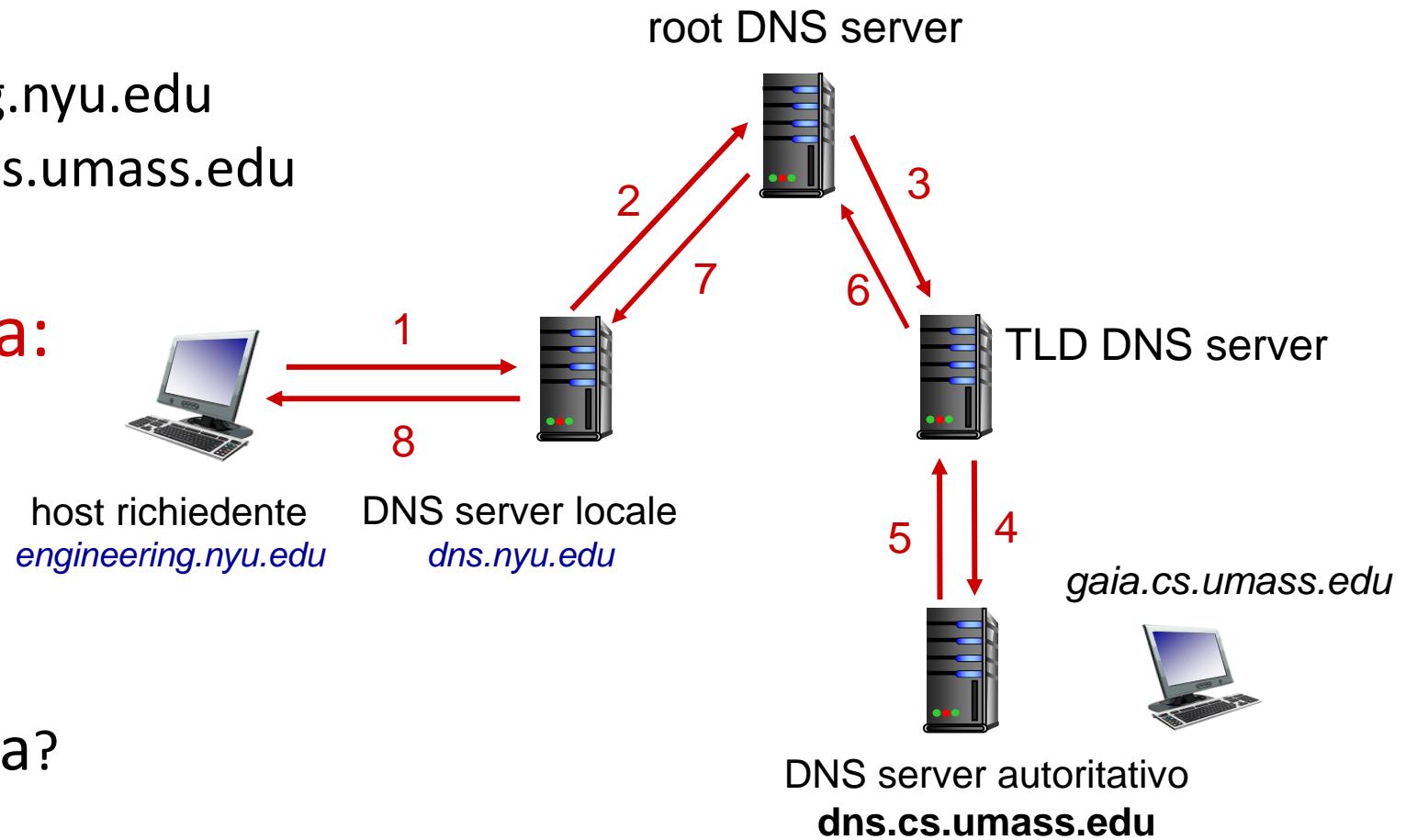


DNS: interrogazione ricorsiva

Esempio: l'host engineering.nyu.edu vuole l'indirizzo IP di gaia.cs.umass.edu

Interrogazione ricorsiva:

- Affida il compito di tradurre il nome al server contattato
- carico pesante ai livelli superiori della gerarchia?



DNS: caching e aggiornamento dei record

- una volta che un (qualsiasi) name server impara la mappatura, la mette nella *cache*, e restituisce *immediatamente* il mapping nella cache in risposta a un query
 - il caching migliora i tempi di risposta
 - le voci della cache vanno in timeout (scompaiono) dopo un certo tempo (TTL)
 - i server TLD sono in genere memorizzati nella cache dei server dei nomi locali
- le voci nella cache potrebbero essere *obsolete*
 - se l'host con nome cambia il suo indirizzo IP, potrebbe non essere conosciuto su Internet fino alla scadenza di tutti i TTL!
 - *traduzione nome->indirizzo best-effort!*

Record DNS

DNS: database distribuito che memorizza 7 record di risorsa (**RR**)

Formato RR: (name, value, type, ttl)

type=A

- name è l'hostname
- value è l'indirizzo IP

type=NS

- name è il dominio (ad esempio, foo.com)
- value è l'hostname dell'autoritative name server per questo dominio

type=CNAME

- name è il nome alias di qualche nome "canonico" (nome vero)
- www.ibm.com è in realtà servereast.backup2.ibm.com
- value è il nome canonico

type=MX

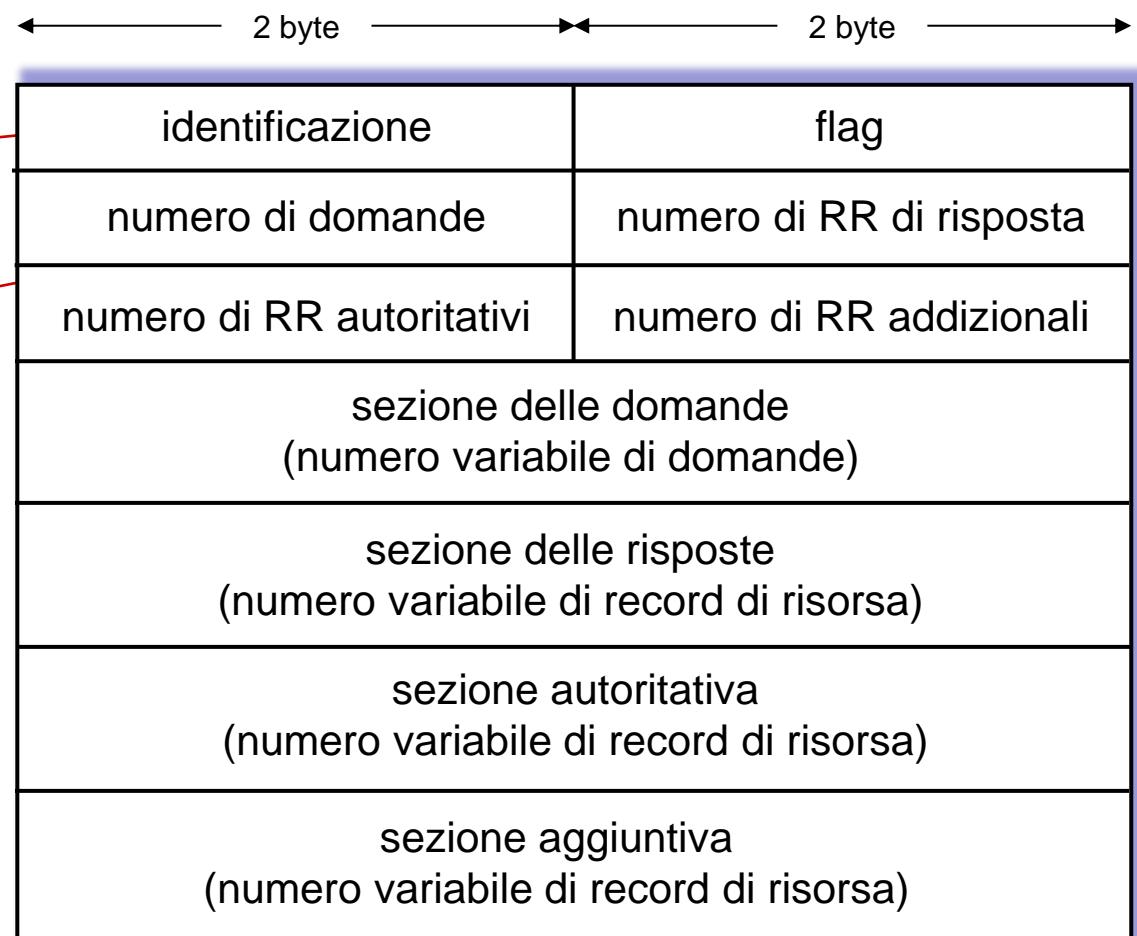
- value è il nome del server di posta associato a name

Messaggi DNS

domande (query) e messaggi di *risposta* (reply), entrambi con lo stesso formato:

Intestazione del messaggio:

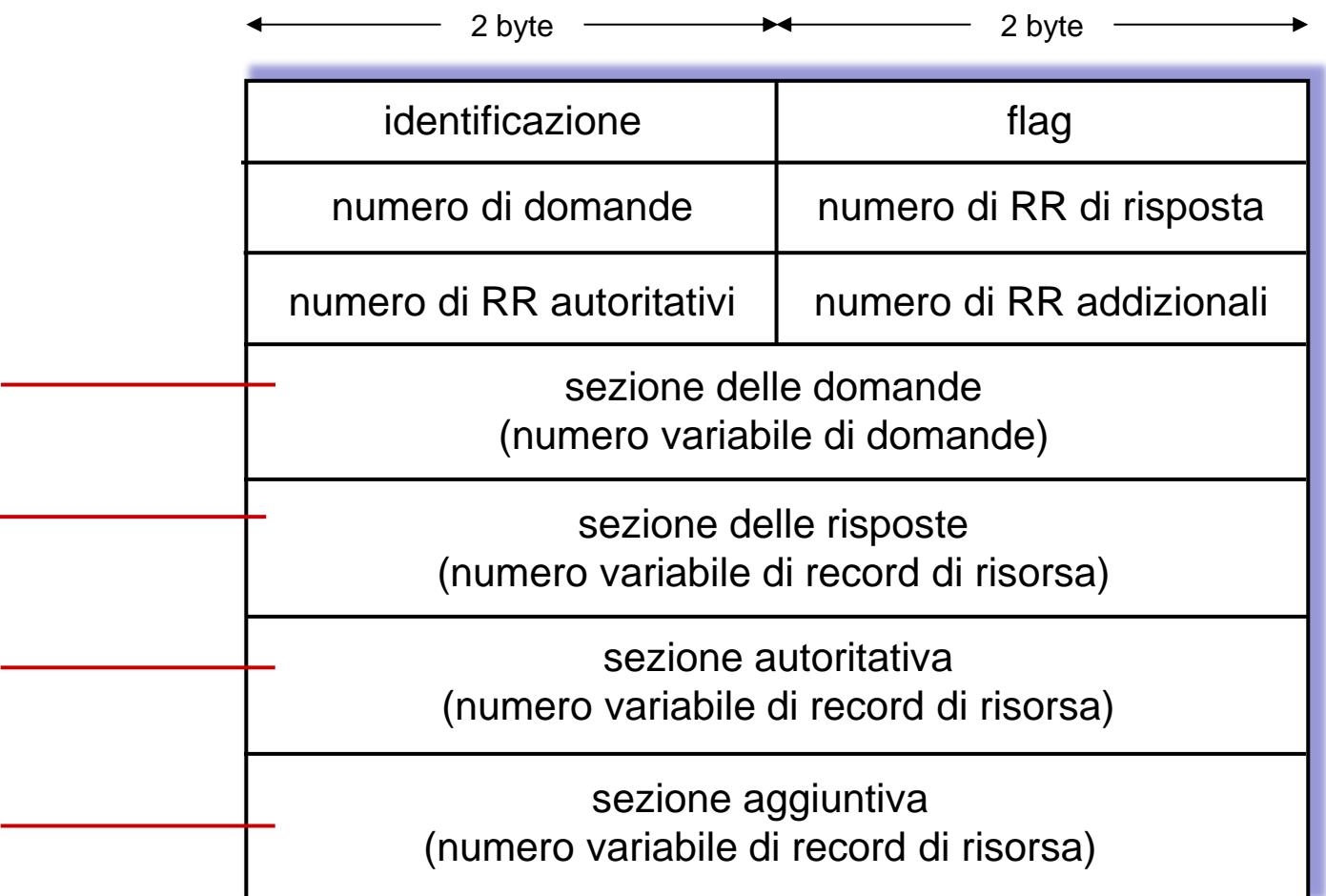
- **identificazione:** numero di 16 bit per la domanda; la risposta alla domanda usa lo stesso numero
- **flag:**
 - domanda o risposta
 - richiesta di ricorsione
 - ricorsione disponibile
 - DNS server autoritativo



Messaggi DNS

domande (query) e messaggi di *risposta* (reply), entrambi con lo stesso formato:

- campi per il nome richiesto e il tipo di domanda
- RR nella risposta alla domanda
- record per i server autoritativi (*referral* verso nameserver di livello più basso; vedi RFC 9471: <https://datatracker.ietf.org/doc/rfc9471/>)
- informazioni extra che possono essere usate (es. se la risposta ad una richiesta di tipo MX contiene un hostname, può essere fornita qui la sua traduzione in IP)



Inserire record nel database DNS

Esempio: abbiamo appena avviato la nuova società “Network Utopia”

- Registriamo il nome **networkuptopia.com** presso il **DNS registrar** (ad esempio, Network Solutions, oppure un altro dei concorrenti accreditati dall'ICANN)
 - forniamo al registrar il nome e gli indirizzi IP degli authoritative name server (primario e secondario)
 - il registrar inserisce due RR nel TLD server .com:
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
- Inseriamo localmente nell'autoritative server
 - un record A per www.networkuptopia.com
 - un record MX per networkutopia.com

Sicurezza del DNS

Attacchi DDoS (distributed denial-of-service)

- bombardare di traffico di root server
 - finora senza successo
 - filtraggio del traffico
 - I server DNS locali mantengono in cache gli indirizzi IP dei server TLD, consentendo di aggirare i root server
- bombardare i server TLD
 - potenzialmente più pericoloso

Attacco di spoofing

- intercettare le query DNS, restituendo risposte fasulle
 - DNS cache poisoning
 - RFC 4033: DNSSEC - servizi di autenticazione

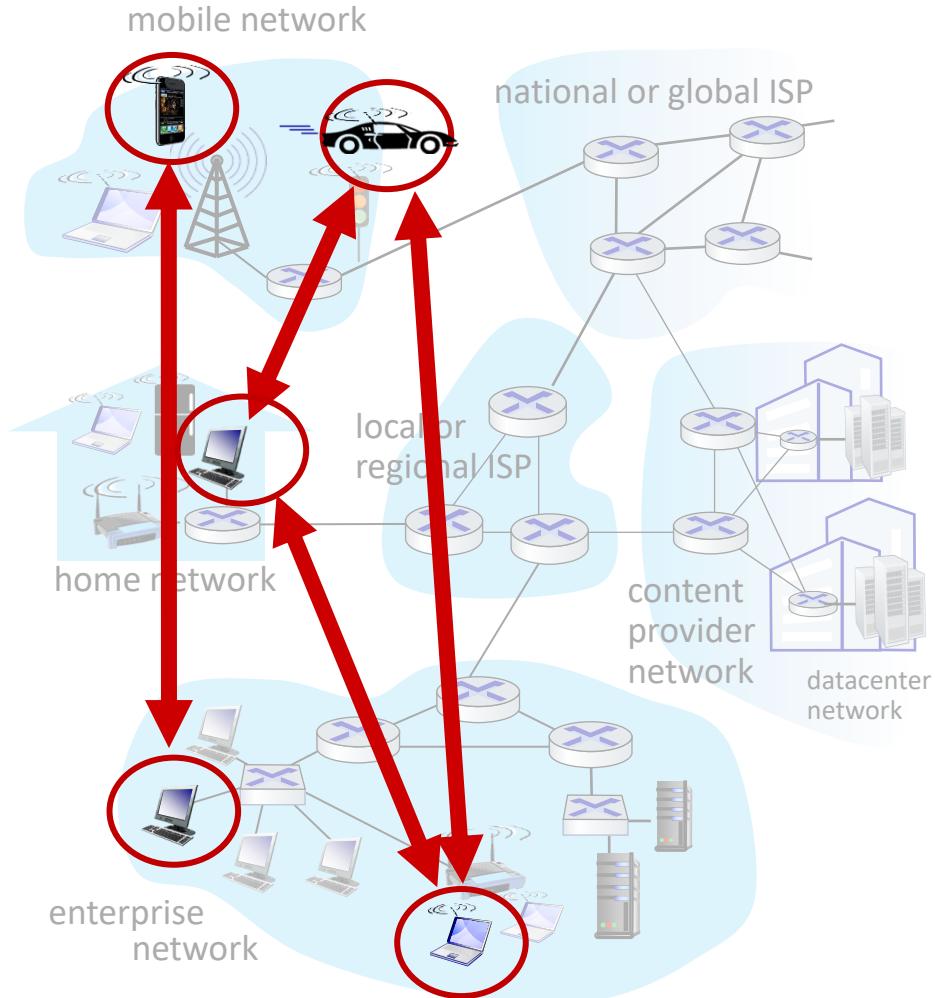
Application Layer: Overview

- Principi delle applicazioni di rete
- Web e HTTP
- E-mail, SMTP, IMAP
- DNS: il servizio di directory di Internet
- Applicazioni P2P
- Streaming video e reti di distribuzione di contenti
- Programmazione delle socket programming con UDP e TCP



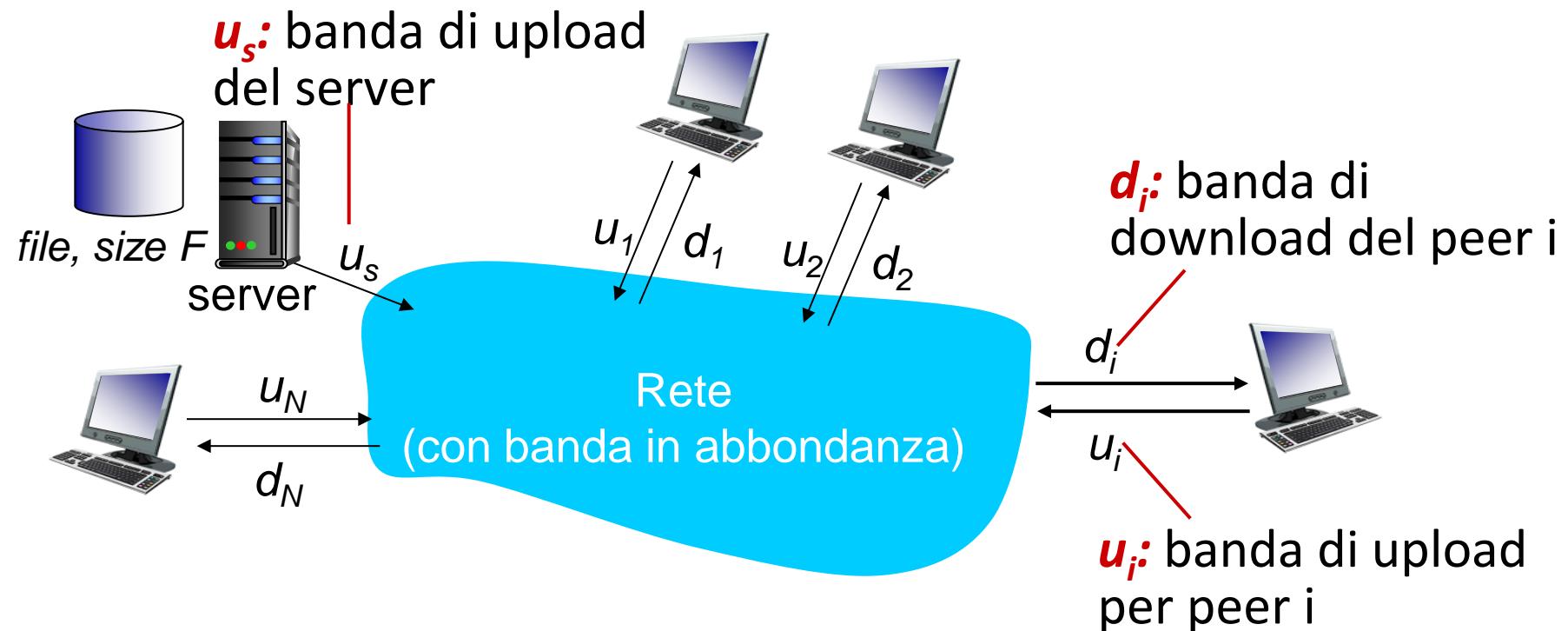
Architettura Peer-to-peer (P2P)

- *nessun server sempre attivo*
- sistemi periferici arbitrari comunicano direttamente
- i peer richiedono un servizio ad altri peer e forniscono un servizio in cambio ad altri peer
 - *scalabilità intrinseca - nuovi peer portano nuova capacità di servizio e nuove richieste di servizio*
- I peer sono connessi a intermittenza e cambiano indirizzo IP
 - gestione complessa
- Esempi: P2P file sharing (BitTorrent), streaming (KanKan), VoIP (Skype)



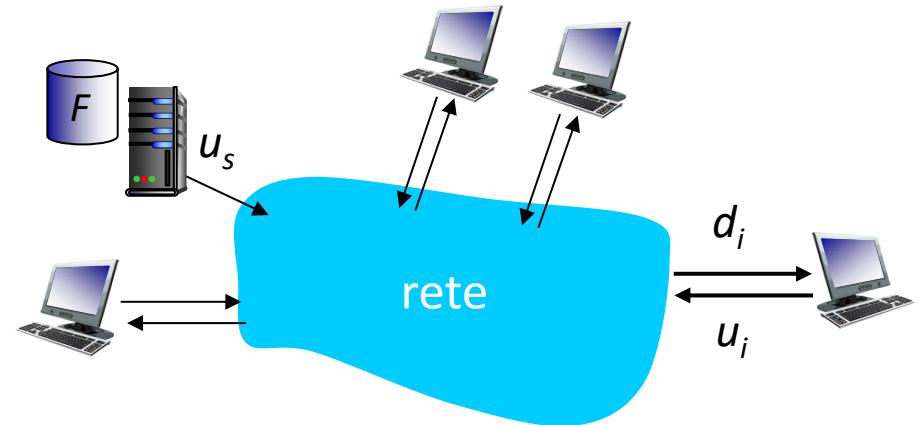
Distribuzione di file: client-server vs P2P

- D:* quanto tempo per distribuire un file (di dimensione F) da un server a N peer?
- la capacità di upload/download dei peer è una risorsa limitata



File distribution time: client-server

- *trasmissione via server*: deve inviare (caricare) in sequenza N copie di file:
 - tempo per inviare una copia: F/u_s
 - tempo per inviare N copie: NF/u_s
- *client*: ogni cliente deve scaricare una copia del file
 - d_{min} = banda di download più bassa
 - tempo di download per il client con banda minima è almeno: F/d_{min}



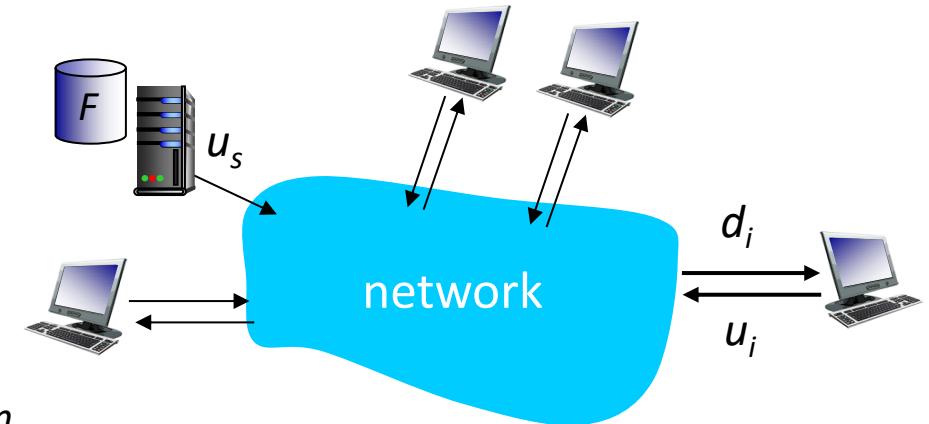
*Tempo per distribuire F
a N client usando
l'approccio client-
server*

$$D_{c-s} \geq \max\{NF/u_s, F/d_{min}\}$$

aumenta linearmente in N

Distribuzione di file: P2P

- *trasmissione via server*: deve trasmettere almeno una copia del file:
 - tempo per inviare un copia: F/u_s
- *client*: ogni cliente deve scaricare una copia del file
 - Tempo per il client più lento, almeno F/d_{min}
- I *client*: come aggregato devono scaricare NF bit
 - capacità totale di upload (che limita la massima velocità di download) è $u_s + \sum u_i$



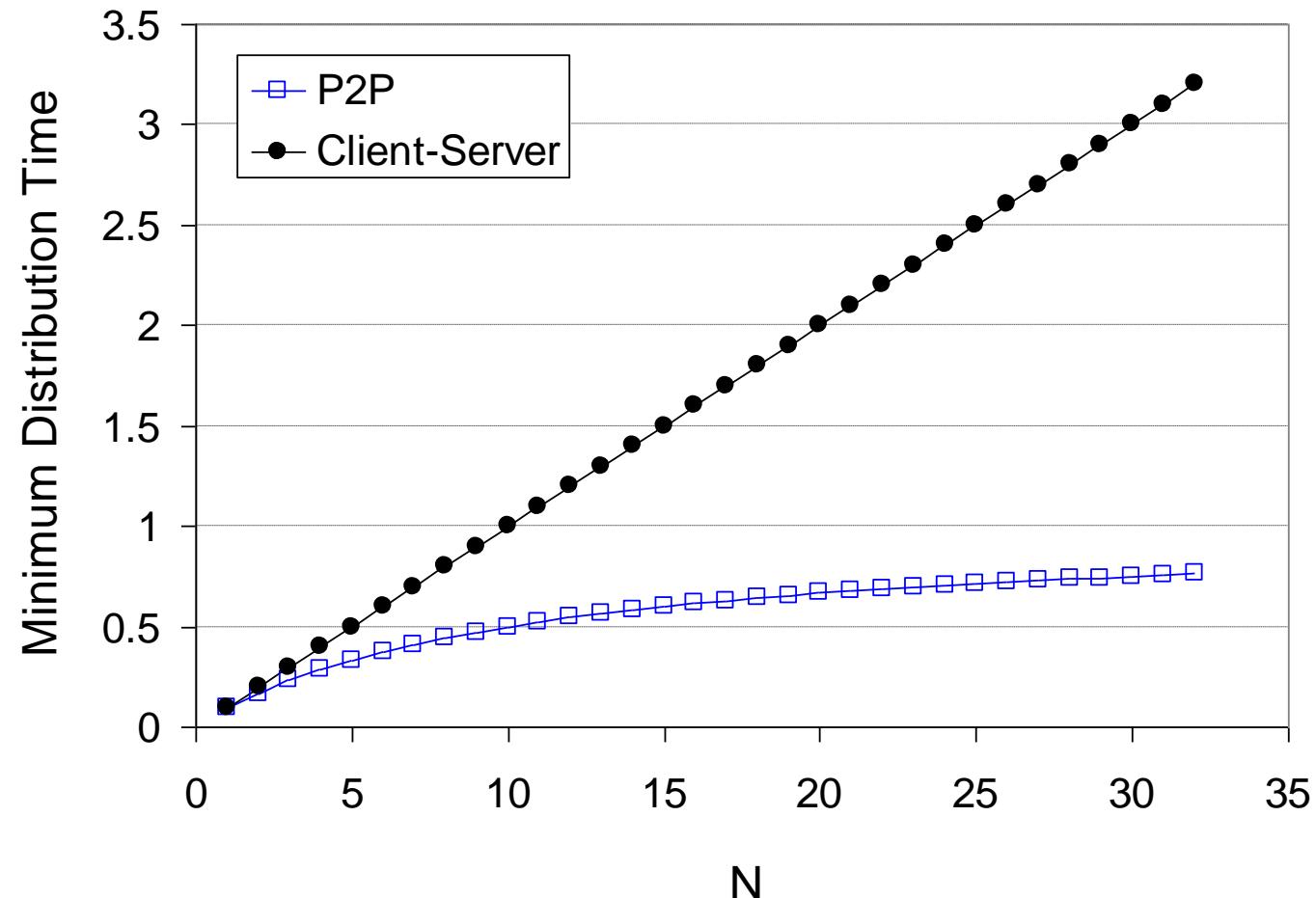
Tempo per distribuire F
a N client usando
l'approccio P2P

$$D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

aumenta linearmente in N ...
... ma anche questo, dato che ogni peer porta con sé la capacità di servizio

Client-server vs. P2P: example

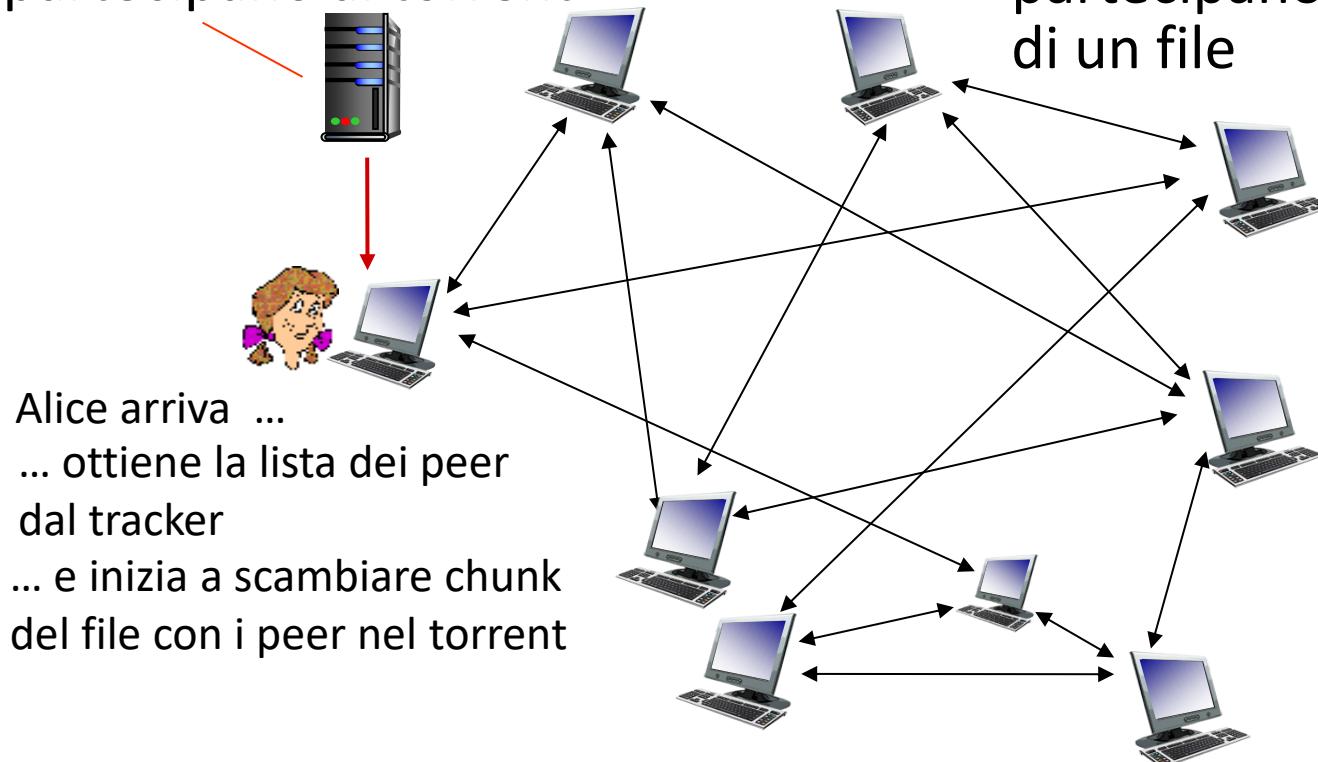
banda di upload del client = u , $F/u = 1$ ora, $u_s = 10u$, $d_{min} \geq u_s$



Distribuzione di file P2P: BitTorrent

- file diviso in chunk (parti), in genere di 256 kB
- i peer nel torrent inviano/ricevono chunk del file

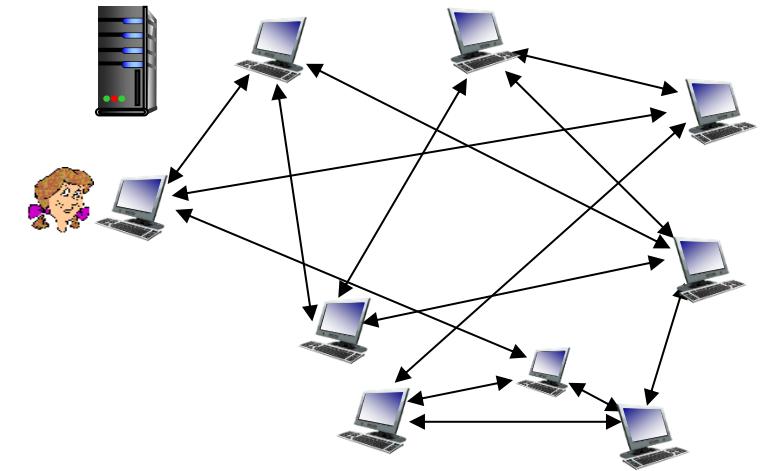
tracker: tiene traccia dei peer che partecipano al torrent



torrent: gruppo di peer che partecipano alla distribuzione di un file

Distribuzione di file P2P: BitTorrent

- Un peer che entra a far parte del torrent:
 - non ha chunk del file, ma li accumulerà nel tempo da altri peer
 - si registra con un tracker, ottenendo la lista di un sottoinsieme dei peer nel torrent (es. 50), stabilisce una connessione con un sottoinsieme di questi, che sono detti peer "vicini" ("neighbors")
 - informa periodicamente il tracker che è ancora nel torrent
- mentre scarica chunk, un peer invia i chunk già in suo possesso agli altri peer
- un peer può cambiare i peer con cui scambia i chunk
- i peer possono andare e venire
- una volta che un peer ha acquisito l'intero file, può (egoisticamente) lasciare il torrent oppure può (altruisticamente) rimanere nel torrent (*come seeder*)



BitTorrent: richiesta e invio di chunk di file

Richiesta di chunk:

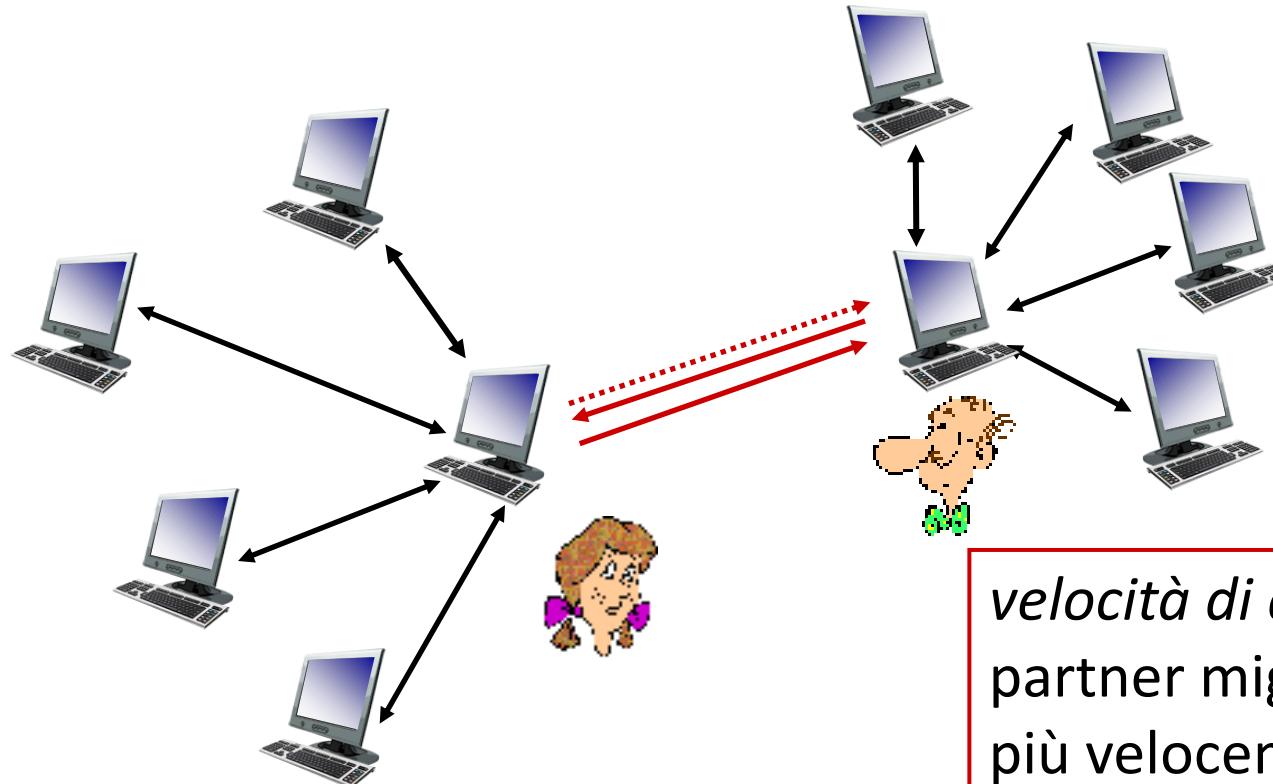
- in ogni momento, peer diversi hanno sottoinsiemi diversi di chunk
- periodicamente, Alice chiede ai peer vicini l'elenco dei chunk in loro possesso
- Alice richiede ai peer i chunk mancanti, adottando la strategia del **rarest first** ("prima i più rari"): uniformando la distribuzione dei chunk, migliora la disponibilità globale e aumenta le possibilità di scambio (maggiore throughput)
- Un peer appena entrato può chiedere un blocco in modo casuale (perché vuole avere il prima possibile un blocco da condividere); mentre, quando sta per completare il file, può adottare la strategia end game e richiedere lo stesso blocco a più peer simultaneamente (cancellando le richieste pendenti appena appena riceve un blocco)

Invio di chunk: tit-for-tat ("pan per focaccia")

- Alice invia i chunk ai quattro peer vicini che attualmente le inviano i chunk *alla velocità più alta*
 - altri peer sono detti choked ("soffocati" o "limitati") (non ricevono chunk da Alice)
 - rivaluta i primi 4 posti ogni 10 secondi
- ogni 30 secondi: seleziona in modo casuale un vicino, inizia a inviare chunk
 - questo peer è detto "optimistically unchoked" ("non limitato/soffocato in maniera ottimistica")
 - il nuovo peer scelto può entrare nella top 4

BitTorrent: tit-for-tat

- (1) Alice scelte Bob come “optimistically unchoked”
- (2) Alice diventa uno dei primi quattro fornitori di Bob; Bob ricambia
- (3) Bob diventa uno dei primi quattro fornitori di Alice.



velocità di caricamento più elevata: trovare partner migliori per gli scambi, ottenere file più velocemente!

Università degli Studi di Roma "Tor Vergata"
Laurea in Informatica

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2023/2024

Livello di applicazione (parte4)

dr. Manuel Fiorelli

manuel.fiorelli@uniroma2.it

<https://art.uniroma2.it/fiorelli>

Basate sulle slide del libro di testo:

https://gaia.cs.umass.edu/kurose_ross/ppt.php

Introduction: 1-103

Livello di applicazione: panoramica

- Principi delle applicazioni di rete
- Web e HTTP
- E-mail, SMTP, IMAP
- DNS: il servizio di directory di Internet
- Applicazioni P2P
- Streaming video e reti di distribuzione di contenti
- Programmazione delle socket programming con UDP e TCP



Streaming video e CDN: contesto

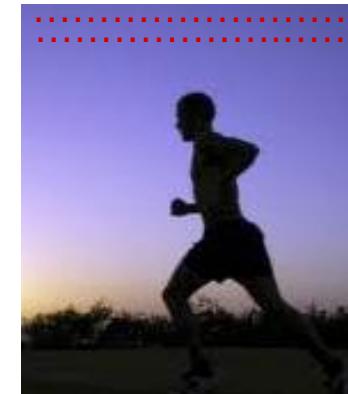
- traffico video in streaming : grande consumatore di larghezza di banda Internet
 - Netflix, YouTube, Amazon Prime: 80% del traffico ISP residenziale (2020)
- *sfida*: scala - come raggiungere ~1B di utenti?
- *sfida*: eterogeneità
 - utenti diversi hanno capacità diverse (ad esempio, cablati o mobili; ricchi di larghezza di banda o poveri di larghezza di banda)
- *soluzione*: infrastruttura distribuita a livello di applicazione



Contenuti multimediali: video

- video: sequenza di immagini visualizzate a tasso costante (*frame rate*)
 - Esempio: 24 immagini al secondo
- immagine digitale: un array di pixel
 - ogni pixel rappresentato da bit
- codifica: utilizzare la ridondanza *all'interno* e *tra* le immagini per ridurre il numero di bit utilizzati per la codifica dell'immagine
 - spaziale (all'interno di una data immagine)
 - temporale (da un'immagine all'altra)

esempio di codifica spaziale:
invece di inviare N valori dello stesso colore (tutti viola), inviare solo due valori: valore del colore (viola) e numero di valori ripetuti (N)



frame *i*

esempio di codifica temporale: invece di inviare il frame completo a $i+1$, invia solo le differenze dal frame *i*

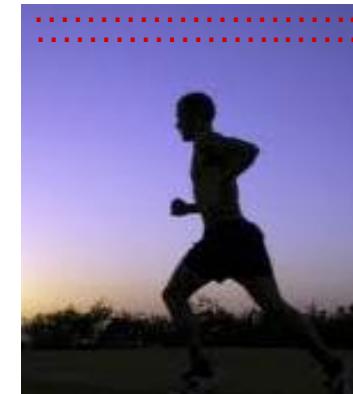


frame *i+1*

Multimedia: video

- **CBR: (constant bit rate):** bit rate costante
- **VBR: (variable bit rate):** bit rate cambia con la quantità di codifica spaziale e temporale
- **esempio:**
 - MPEG-1 (CD-ROM) 1.5 Mbps
 - MPEG-2 (DVD) 3-6 Mbps
 - MPEG-4 (spesso usato in Internet, 64kbps – 12 Mbps)

esempio di codifica spaziale:
invece di inviare N valori dello stesso colore (tutti viola), inviare solo due valori: valore del colore (viola) e numero di valori ripetuti (N)



frame *i*

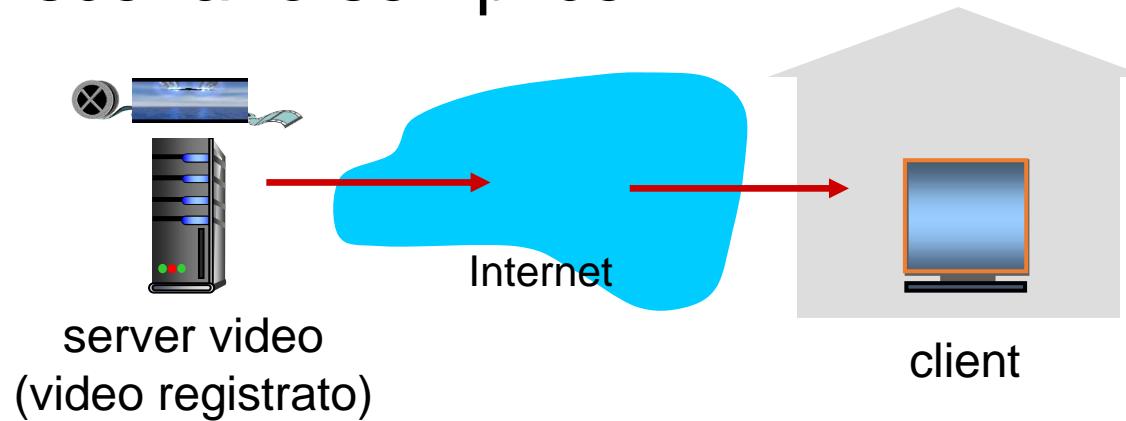
esempio di codifica temporale: invece di inviare il frame completo a $i+1$, invia solo le differenze dal frame *i*



frame *i+1*

Streaming video di contenuti registrati

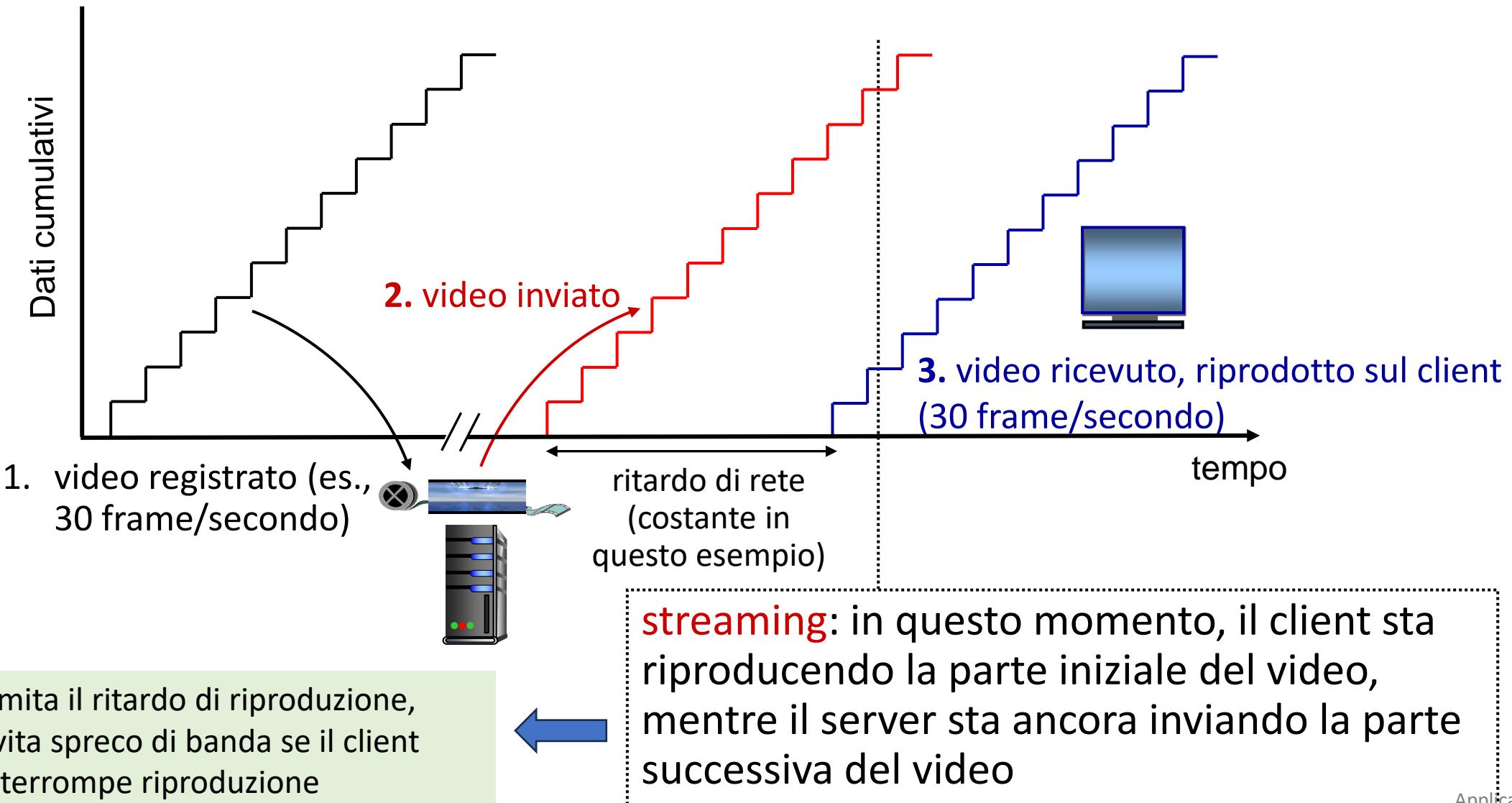
scenario semplice



Sfide principali:

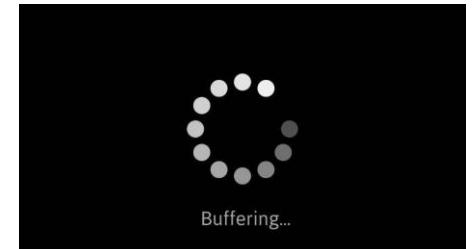
- la larghezza di banda da server a client varia nel tempo, con il variare dei livelli di congestione della rete (rete residenziale, rete di accesso, nucleo della rete, server video)
- la perdita di pacchetti, i ritardi dovuti alla congestione ritardano la riproduzione o comportano una scarsa qualità video.

Streaming video di contenuti registrati

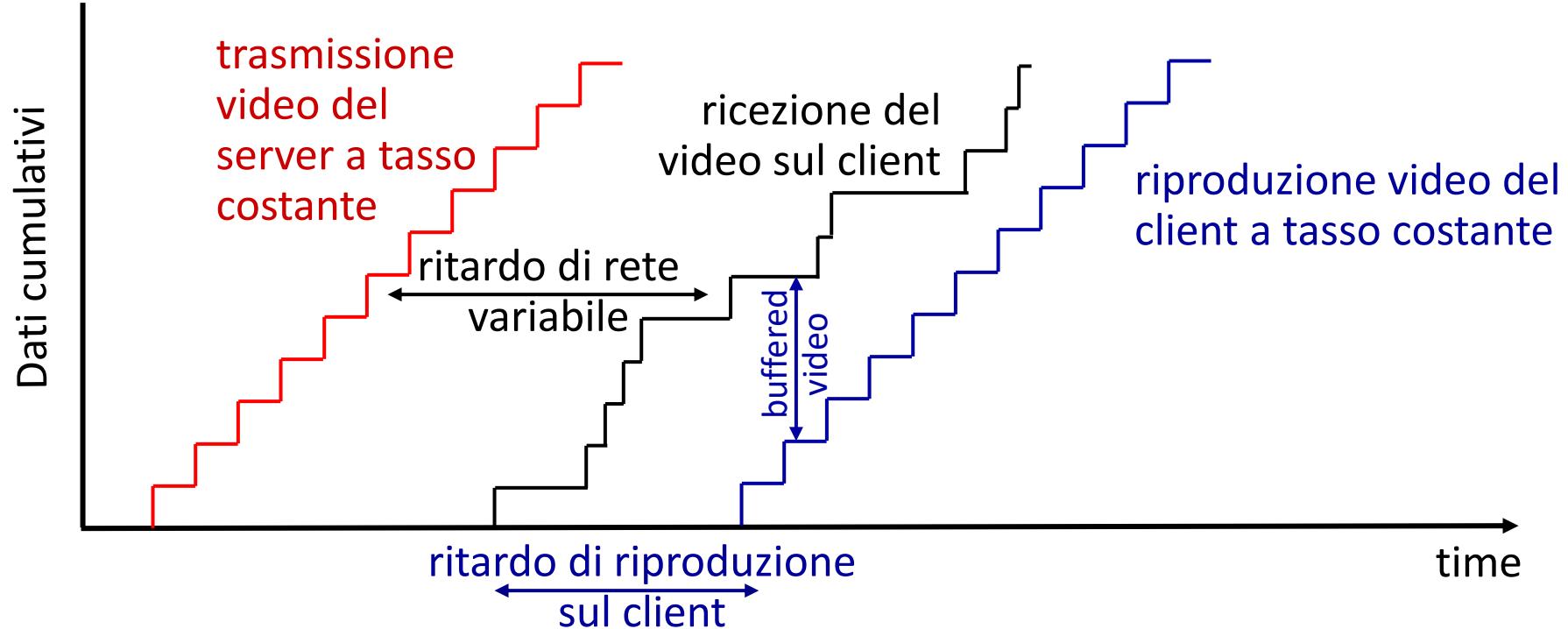


Streaming video di contenuti registrati

- **vincolo di riproduzione continua:** quando la riproduzione inizia, dovrebbe procedere secondo i tempi di registrazione originali
 - ... ma i ritardi di rete sono variabili (jitter), quindi avrà bisogno di un buffer lato client per soddisfare i vincoli di riproduzione continua
- altre sfide:
 - interattività del client: pausa, avanzamento veloce, riavvolgimento, salti attraverso il video
 - i pacchetti video possono essere persi, ritrasmessi



Streaming video di contenuti registrati



- *buffering lato client e ritardo di riproduzione*: compensare il ritardo aggiunto dalla rete, il jitter (variazione) del ritardo

Streaming video di contenuti registrati

■ Streaming UDP:

- Il server invia pacchetti video in modo da eguagliare il bit rate del video stesso, traendo vantaggio dall'assenza di controllo della congestione (es. bit rate 2 Mbps e pacchetti da 8000 bit → il server invia un pacchetto ogni 4 ms)
- Buffer lato client equivalente a pochi secondi di video
- Connessione di controllo separata attraverso il quale il client può inviare comandi, quali pausa, salto, etc.
- Incapace di rispettare il vincolo di riproduzione continua, se la banda disponibile scende periodicamente sotto al bit rate del video (che è il tasso di consumo)

■ Streaming HTTP

- Il server trasmette alla massima velocità consentita (es. dal controllo della congestione): se la velocità di ricezione è maggiore del bit rate del video, il buffer del client continua a crescere anche durante la riproduzione (*prefetching*) fino al riempimento, dopodiché il controllo del flusso limiterà il tasso di trasmissione al tasso di consumo del client
- può assorbire fluttuazioni del throughput, finché il throughput medio è maggiore del bit rate
- Riga di intestazione *Range* nelle richieste per saltare

■ streaming dinamico adattativo su HTTP

- Consente di scegliere tra versioni con livelli di qualità differenti anche durante la riproduzione

Streaming multimediale: DASH

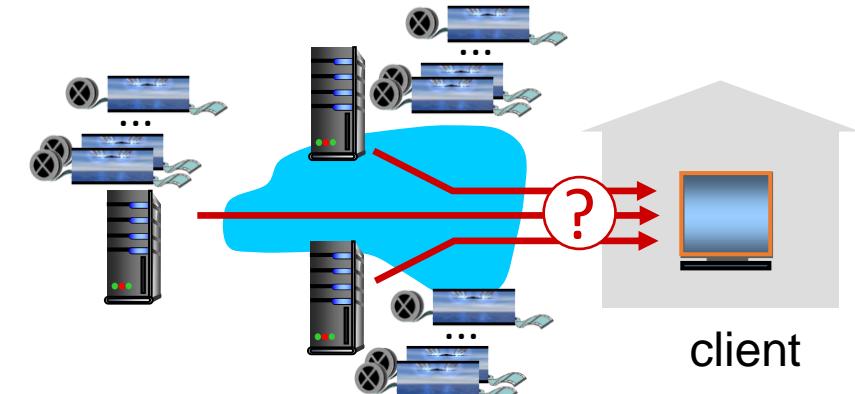
Dynamic, Adaptive
Streaming over HTTP

server:

- divide il file video in più chunk
- ogni chunk è codificata in più versioni, con bit rate differenti
- versioni diverse sono memorizzate in file diversi
- i file sono replicati in vari nodi CDN
- *manifest file (file manifesto)*: fornisce gli URL per i diversi chunk

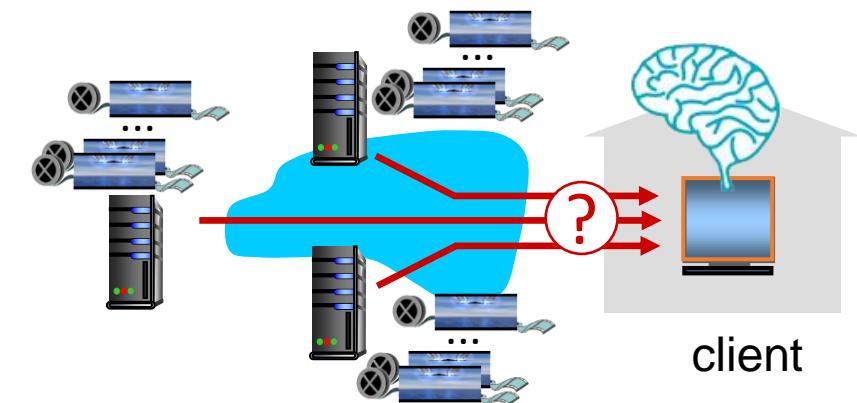
client:

- stima periodicamente la banda da server a client
- consultando il manifesto, richiede un chunk alla volta
 - sceglie versione con il bit rate più alto sostenibile data la larghezza di banda corrente
 - può scegliere versioni con bit rate differenti in momenti diversi (a seconda della larghezza di banda disponibile in quel momento), e da server diversi



Streaming multimediale: DASH

- “*intelligenza*” sul client: il client determina
 - *quando* richiedere un chunk (in modo che non si verifichi la starvation del buffer o l'overflow)
 - *che encoding rate richiedere (qualità più alta quando c'è più larghezza di banda)*
 - *dove* richiedere il chunk (può richiedere dal server che è "vicino" al client o ha banda larga)



Streaming video = codifica + DASH + buffering di riproduzione

Reti per la distribuzione di contenuti - Content distribution networks (CDNs)

sfida: come trasmettere contenuti in streaming (selezionati tra milioni di video) a centinaia di migliaia di utenti simultanei?

- *opzione 1:* unico, enorme data center
 - singolo punto di rottura (single point of failure)
 - punto di congestione della rete
 - percorso lungo (e possibilmente congestionato) verso i clienti lontani

.... molto semplicemente: questa soluzione *non è scalabile*

Reti per la distribuzione di contenuti - Content distribution networks (CDNs)

sfida: come trasmettere contenuti in streaming (selezionati tra milioni di video) a centinaia di migliaia di utenti simultanei?

- *opzione 2:* memorizzare/servire più copie di video in più siti geograficamente distribuiti (*CDN*)
 - *enter deep (entrare in profondità):* installare server della CDN in profondità dentro molte reti di accesso
 - essere vicini agli utenti -> minore ritardi e maggior throughput, ma maggiore complessità di gestione e manutenzione
 - Akamai: 240,000 server disposti in > 120 paesi (2015)
 - *bring home (portare a casa):* pochi grandi cluster (decine, per esempio) in IXP vicino alle reti di accesso
 - usato da Limelight



Akamai oggi:



The Akamai Edge Today

360K
servers

100+
million hits
per second

7+
trillion
deliveries
per day

175+
terabits per
second
(250+ peak)

4,200+
locations

1,350+
networks

840+
cities

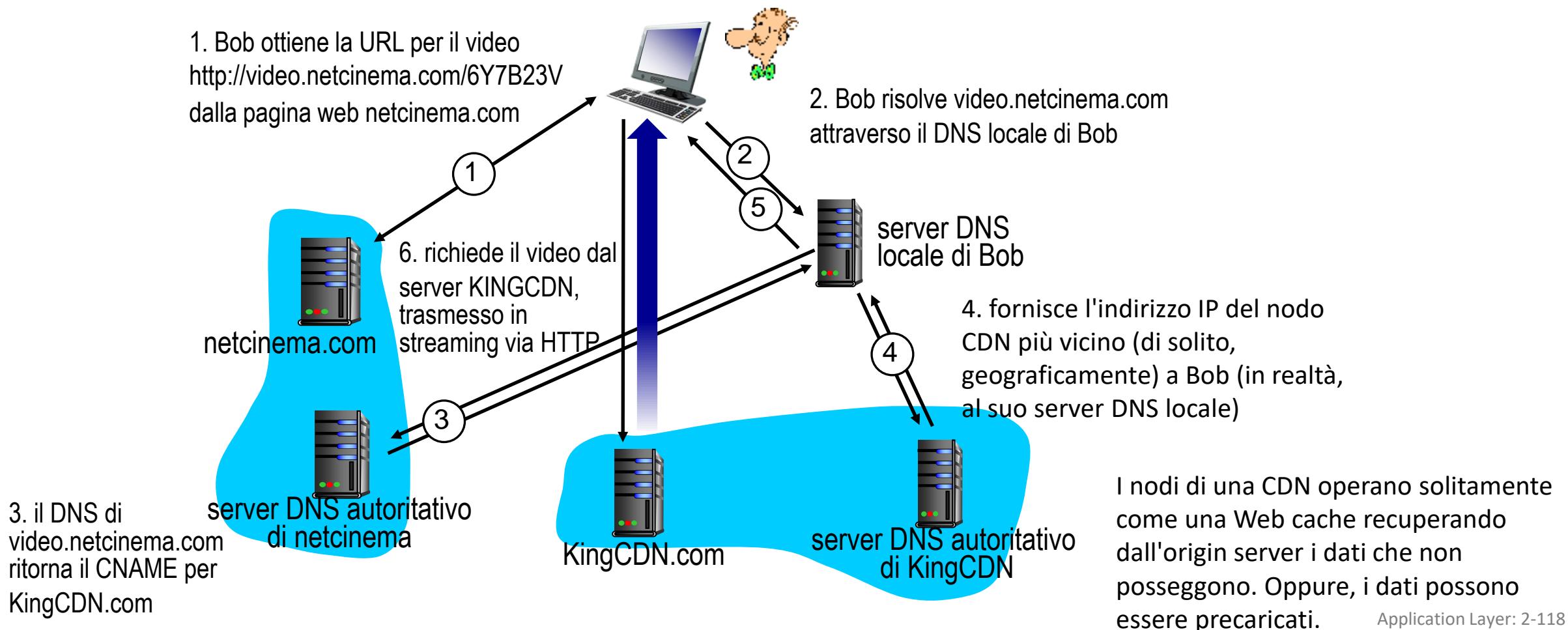
135
countries

Fonte: <https://networkingchannel.eu/living-on-the-edge-for-a-quarter-century-an-akamai-retrospective-downloads/>

Accesso ai contenuti CDN: uno sguardo da vicino

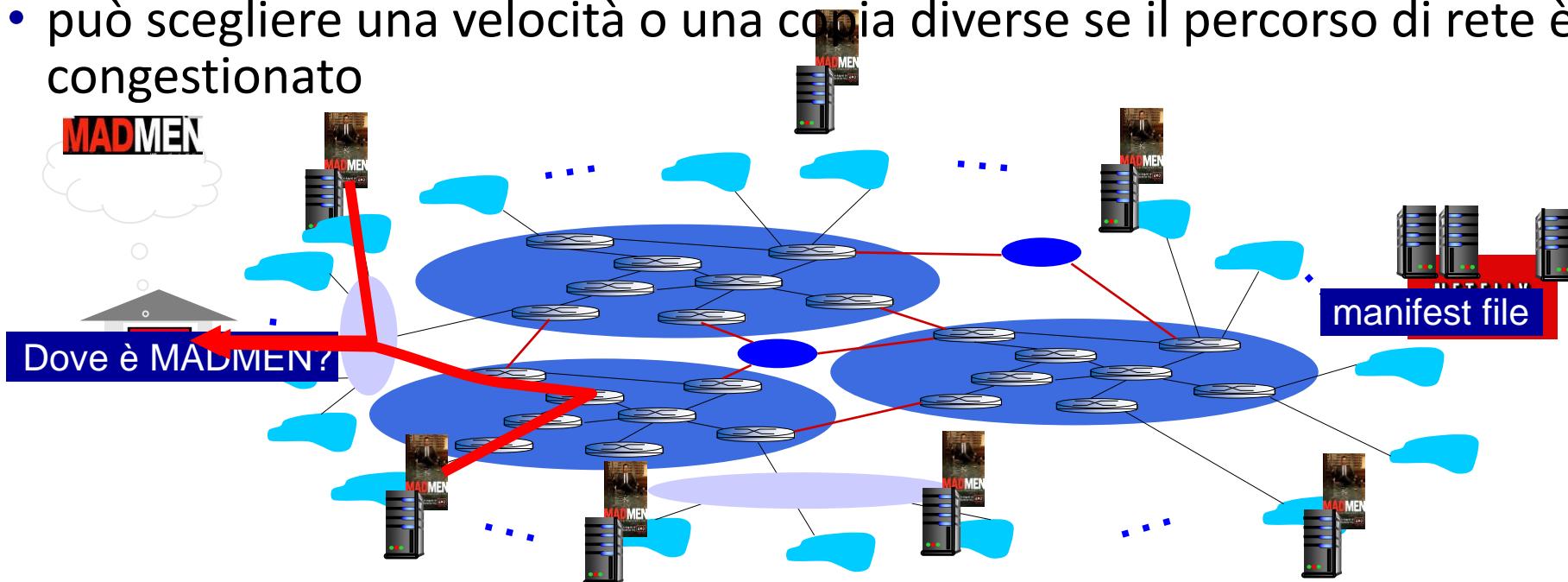
Bob (client) richiede il video <http://video.netcinema.com/6Y7B23V>

- video memorizzato sulla CDN a <http://netcinema.KingCDN.com/6Y7B23V>

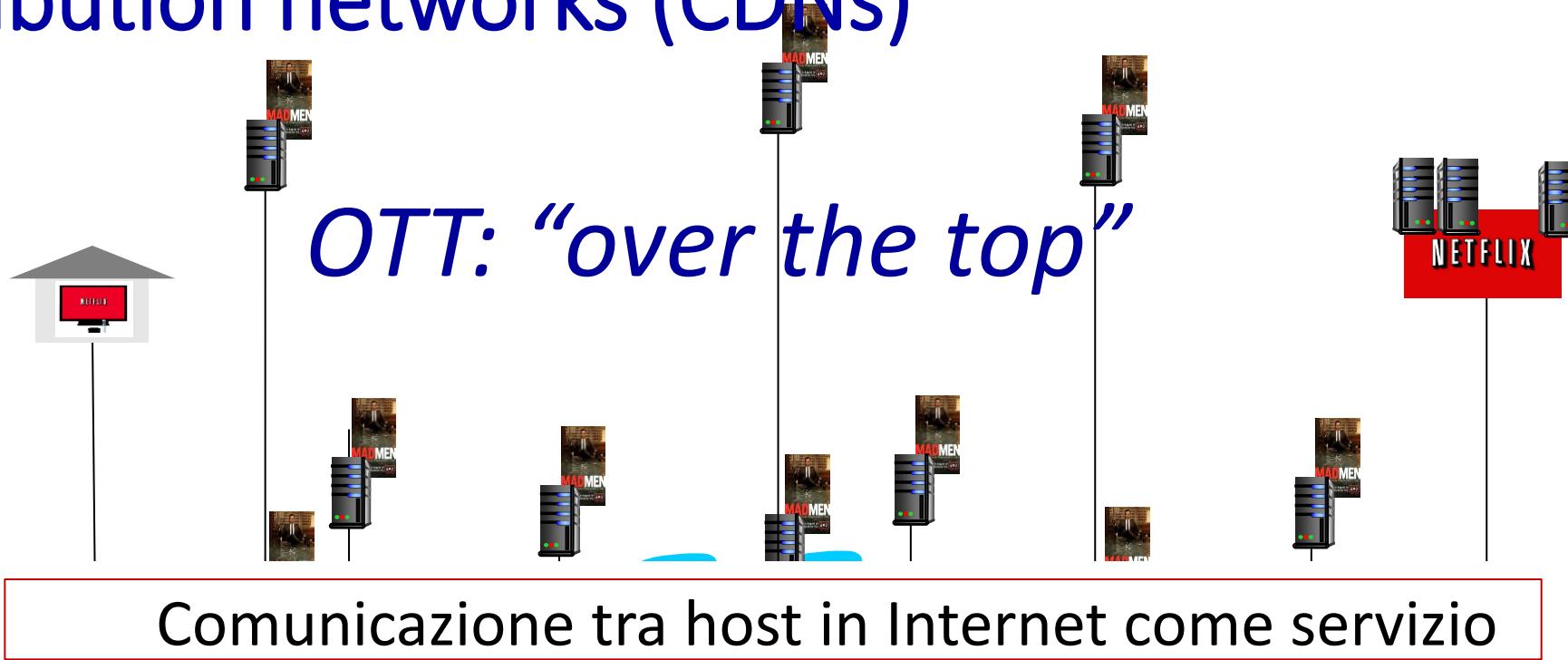


Come funziona Netflix?

- Netflix: memorizza copie dei contenuti (ad esempio, MADMEN) sui nodi (in tutto il mondo) della sua CDN OpenConnect
- l'abbonato richiede il contenuto, il fornitore di servizi restituisce il manifesto
 - utilizzando il manifest, il client recupera i contenuti alla massima velocità supportabile
 - può scegliere una velocità o una copia diverse se il percorso di rete è congestionato



Reti per la distribuzione di contenuti - Content distribution networks (CDNs)



Sfide OTT: affrontare un Internet congestionato dal "bordo"

- quale contenuto inserire in quale nodo CDN?
- da quale nodo CDN recuperare i contenuti? A quale velocità?

Livello di applicazione: panoramica

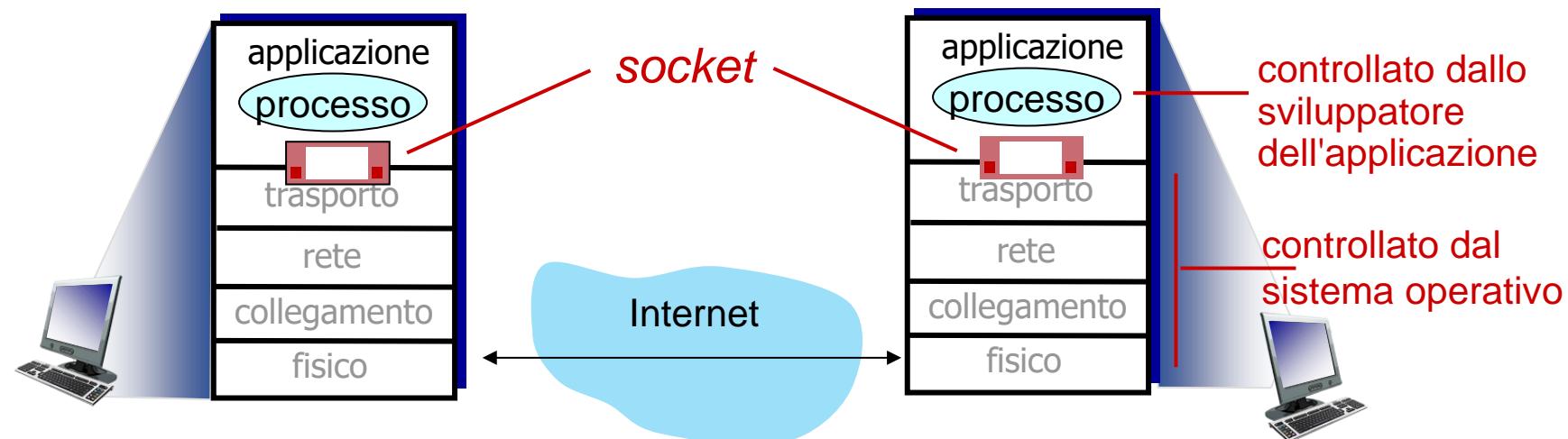
- Principi delle applicazioni di rete
- Web e HTTP
- E-mail, SMTP, IMAP
- DNS: il servizio di directory di Internet
- Applicazioni P2P
- Streaming video e reti di distribuzione di contenti
- Programmazione delle socket programming con UDP e TCP



Programmazione delle socket

goal: imparare a costruire applicazioni client/server che comunicano utilizzando le socket

socket: porta tra il processo applicativo e il protocollo di trasporto



Programmazione delle socket

Due tipi di socket per due servizi di trasporto:

- *UDP*: datagramma inaffidabile
- *TCP*: affidabile, orientato agli stream di byte

Applicazione d'Esempio:

1. il client legge una riga di caratteri (dati) dalla sua tastiera e la invia al server
2. il server riceve i dati e converte i caratteri in maiuscolo
3. il server invia i dati modificati al client
4. il client riceve i dati modificati e li visualizza sul proprio schermo

Socket programming with UDP

UDP: no “connection” between client and server:

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server processes

Client/server socket interaction: UDP



server (running on serverIP)

```
create socket, port= x:  
serverSocket =  
socket(AF_INET,SOCK_DGRAM)
```

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number



client

```
create socket:  
clientSocket =  
socket(AF_INET,SOCK_DGRAM)
```

Create datagram with serverIP address
And port=x; send datagram via
clientSocket

read datagram from
clientSocket
close
clientSocket

Example app: UDP client

Python UDPCClient

```
include Python's socket library → from socket import *
serverName = 'hostname'
serverPort = 12000
create UDP socket → clientSocket = socket(AF_INET,
                                             SOCK_DGRAM)
get user keyboard input → message = input('Input lowercase sentence:')
attach server name, port to message; send into socket → clientSocket.sendto(message.encode(),
                           (serverName, serverPort))
read reply data (bytes) from socket → modifiedMessage, serverAddress =
                                         clientSocket.recvfrom(2048)
print out received string and close socket → print(modifiedMessage.decode())
                                                clientSocket.close()
```

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port number 12000 → serverSocket.bind(('', serverPort))
                                         print('The server is ready to receive')
loop forever → while True:
Read from UDP socket into message, getting →   message, clientAddress = serverSocket.recvfrom(2048)
client's address (client IP and port)           modifiedMessage = message.decode().upper()
                                               serverSocket.sendto(modifiedMessage.encode(),
send upper case string back to this client →   clientAddress)
```

Socket programming with TCP

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

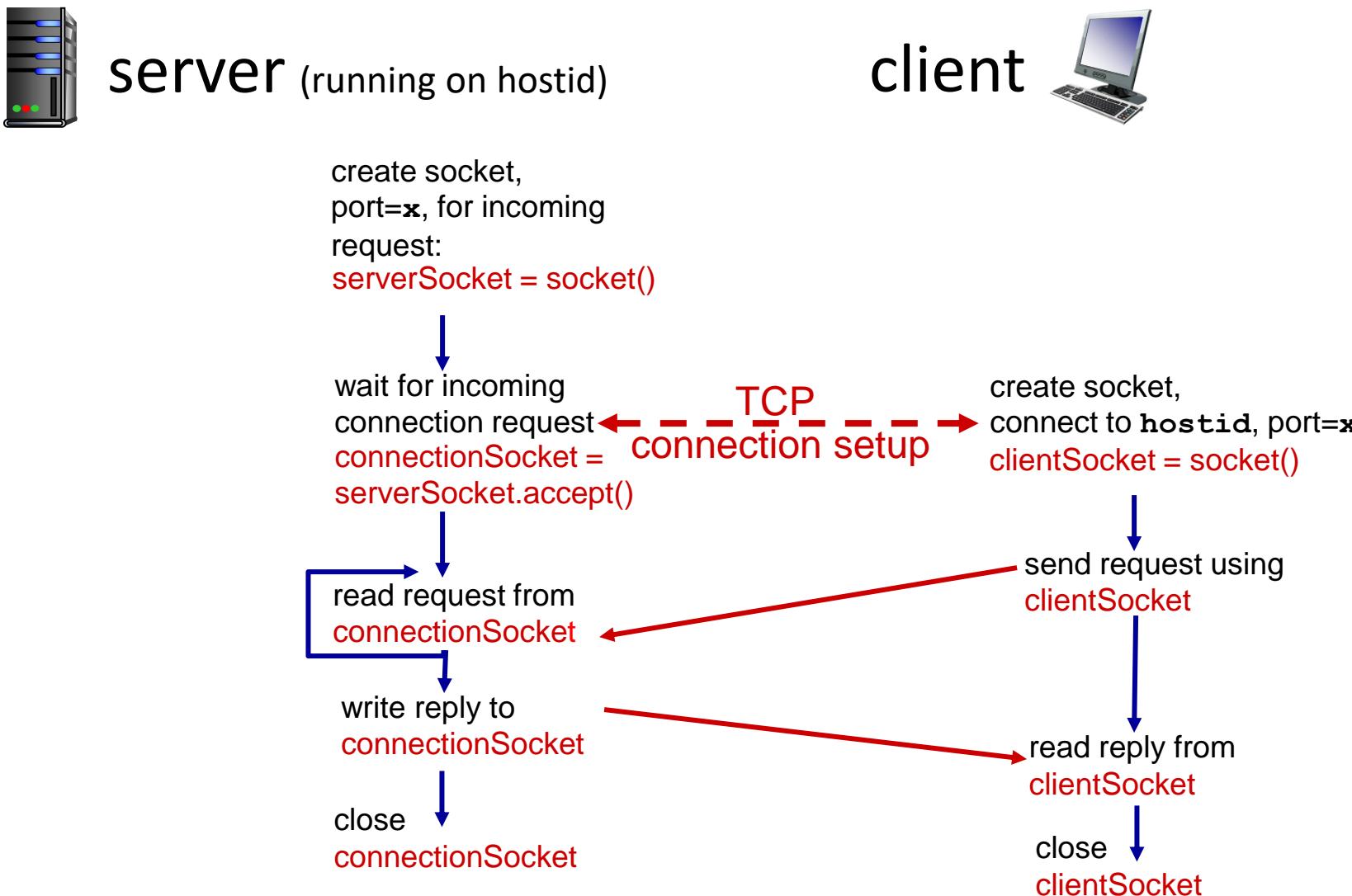
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket*: client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
 - allows server to talk with multiple clients
 - client source port # and IP address used to distinguish clients (more in Chap 3)

Application viewpoint

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server processes

Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

create TCP socket for server,
remote port 12000 → clientSocket = socket(AF_INET, SOCK_STREAM)

No need to attach server name, port → modifiedSentence = clientSocket.recv(1024)

Example app: TCP server

Python TCP Server

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(("",serverPort))
serverSocket.listen(1)
print('The server is ready to receive')
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
connectionSocket.close()
```

create TCP welcoming socket → from socket import *

server begins listening for incoming TCP requests → serverPort = 12000
→ serverSocket = socket(AF_INET,SOCK_STREAM)
→ serverSocket.bind(("",serverPort))
→ serverSocket.listen(1)

loop forever → print('The server is ready to receive')

server waits on accept() for incoming requests, new socket created on return → while True:

read bytes from socket (but not address as in UDP) → connectionSocket, addr = serverSocket.accept()
→ sentence = connectionSocket.recv(1024).decode()
→ capitalizedSentence = sentence.upper()
→ connectionSocket.send(capitalizedSentence.
→ encode())

close connection to this client (but *not* welcoming socket) → connectionSocket.close()

Capitolo 2: riassunto

Io studio del livello applicativo della rete è ora completo!

- architetture delle applicazioni
 - client-server
 - P2P
- Requisiti di servizio delle applicazioni:
 - affidabilità, larghezza di banda, ritardo
- Modello di servizio di trasporto di Internet
 - orientato alla connessione, affidabile: TCP
 - non affidabile, datagrammi: UDP
- specifici protocolli:
 - HTTP
 - SMTP, IMAP
 - DNS
 - P2P: BitTorrent
- streaming video, CDN
- programmazione delle socket:
 - socket TCP, UDP

Capitolo 2: riassunto

La cosa più importante è che abbiamo imparato a conoscere i *protocolli!*

- tipico scambio di messaggi di richiesta e risposta
 - il client richiede informazioni o servizi
 - il server risponde con dati, codici di stato
- formato dei messaggi:
 - *intestazione*: campi che informazioni informazioni sui dati
 - *dati*: informazione (carico) trasportata

temi importanti:

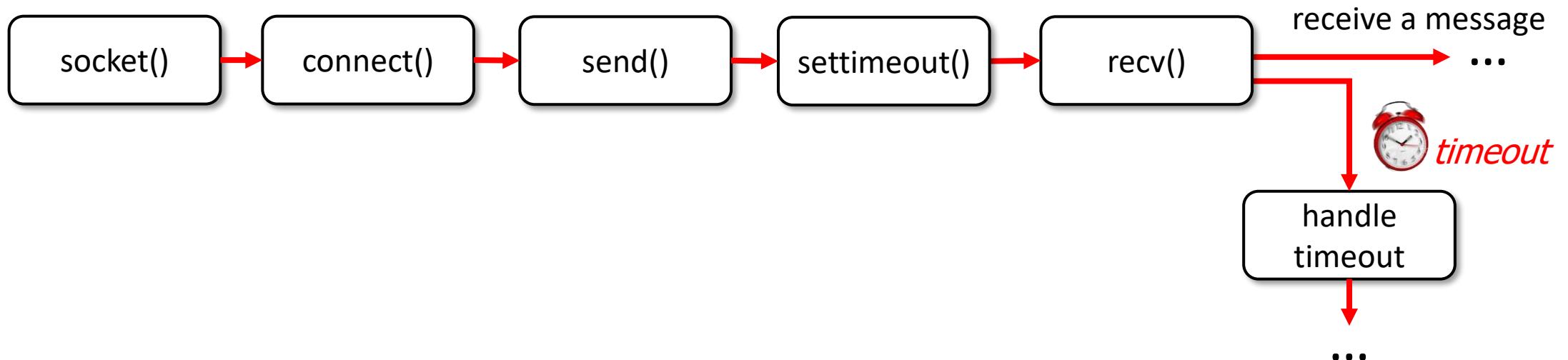
- centralizzato vs. decentralizzato
- senza stato vs. con stato
- scalabilità
- Trasferimento di messaggi affidabile e non
- “complessità ai margini della rete”

Additional Chapter 2 slides

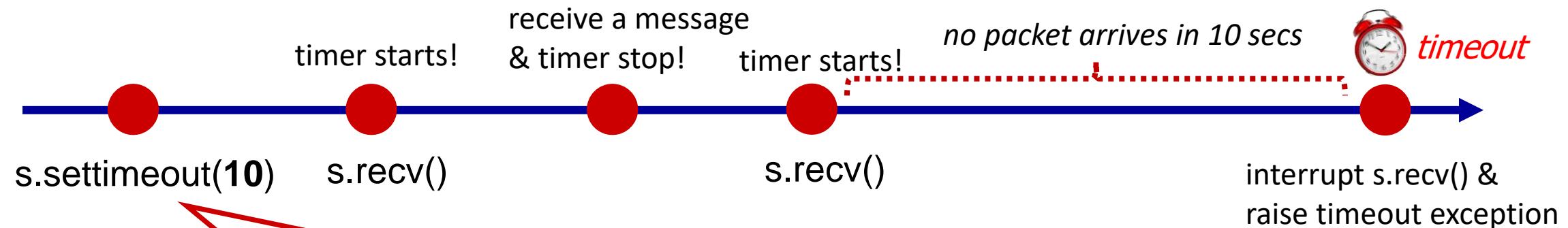
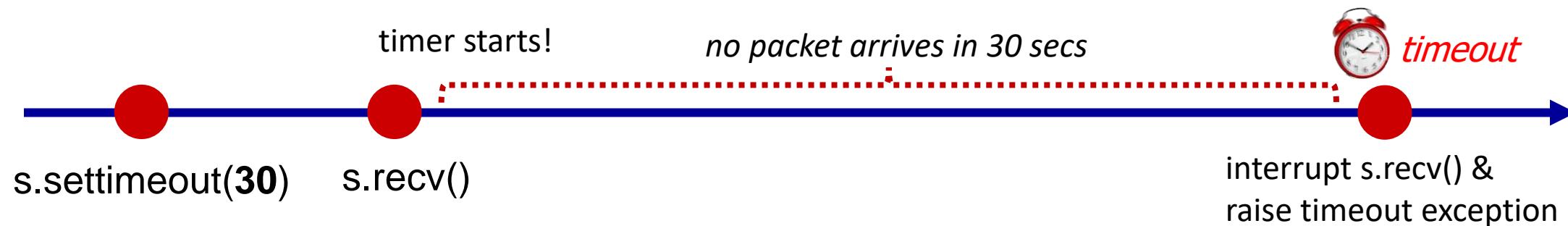
JFK note: the timeout slides are important IMHO if one is doing a programming assignment (especially an RDT programming assignment in Chapter 3), since students will need to use timers in their code, and the TRY/EXCEPT is really the easiest way to do this. I introduce this here in Chapter 2 with the socket programming assignment since it teaches something (how to handle exceptions/timeouts), and lets students learn/practice that before doing the RDT programming assignment, which is harder

Socket programming: waiting for multiple events

- sometimes a program must **wait for one of several events** to happen, e.g.,:
 - wait for either (i) a reply from another end of the socket, or (ii) timeout: **timer**
 - wait for replies from several different open sockets: **select()**, multithreading
- timeouts are used extensively in networking
- using timeouts with Python socket:



How Python socket.settimeout() works?



Set a timeout on all future socket operations of that specific socket!

Python try-except block

Execute a block of code, and handle “exceptions” that may occur when executing that block of code

try:

 <do something>

except <exception>:

 <handle the exception>

Executing this **try code block** may cause exception(s) to catch. If an exception is raised, execution jumps directly into **except code block**

this **except code block** is only executed *if an <exception> occurred* in the **try code block** (note: except block is *required* with a try block)

Socket programming: socket timeouts

Toy Example:



- A shepherd boy tends his master's sheep.
- If he sees a wolf, he can send a message to villagers for help using a TCP socket.
- The boy found it fun to connect to the server without sending any messages. But the villagers don't think so.
- And they decided that if the boy connects to the server and doesn't send the wolf location **within 10 seconds for three times**, they will **stop listening** to him forever and ever.

set a 10-seconds timeout on
all future socket operations

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(("",serverPort))
serverSocket.listen(1)
```

counter = 0

while counter < 3:

```
    connectionSocket, addr = serverSocket.accept()
    connectionSocket.settimeout(10)
```

try:

```
    wolf_location = connectionSocket.recv(1024).decode()
    send_hunter(wolf_location) # a villager function
    connectionSocket.send('hunter sent')
```

except timeout:

```
    counter += 1
```

```
connectionSocket.close()
```

timer starts when recv() is called and will
raise timeout exception if there is no
message within 10 seconds.

catch socket timeout exception

Sample SMTP interaction

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```