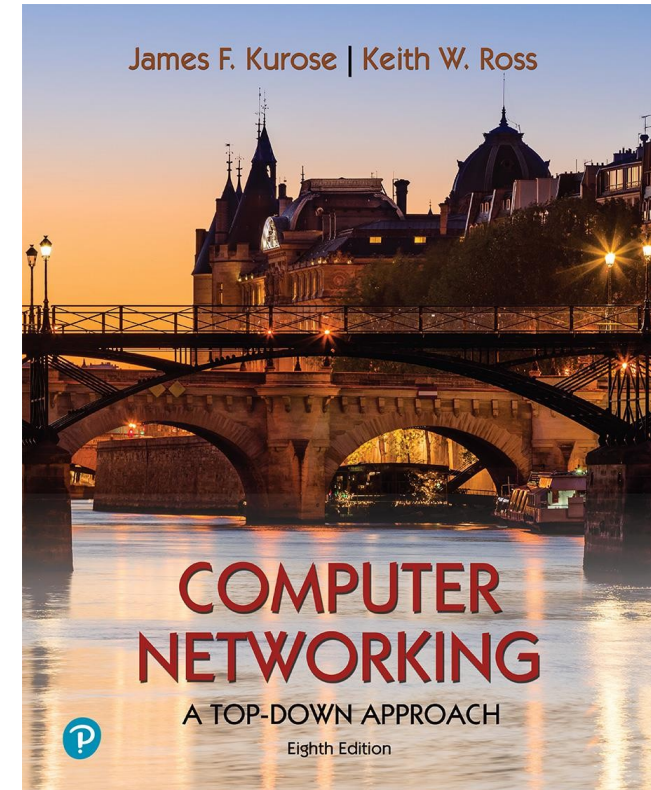


Capitolo 2 Livello applicativo



Reti informatiche: un approccio dall'alto verso il basso

8^aedizione
Jim Kurose, Keith Ross Pearson, 2020

Livello applicativo: panoramica

- Principi delle applicazioni di rete
- Web e HTTP
- E-mail, SMTP, IMAP
- Il sistema dei nomi di dominio DNS
- Applicazioni P2P
- Streaming video e reti di distribuzione dei contenuti
- Programmazione socket con UDP e TCP



Alcune applicazioni di rete

- social network
- Web
- messaggistica di testo
- e-mail
- giochi di rete multiutente
- streaming di video archiviati (YouTube, Hulu, Netflix)
- condivisione file P2P
- voce su IP (ad es. Skype)
- videoconferenze in tempo reale (ad es. Zoom)
- ricerca su Internet
- accesso remoto
- ...

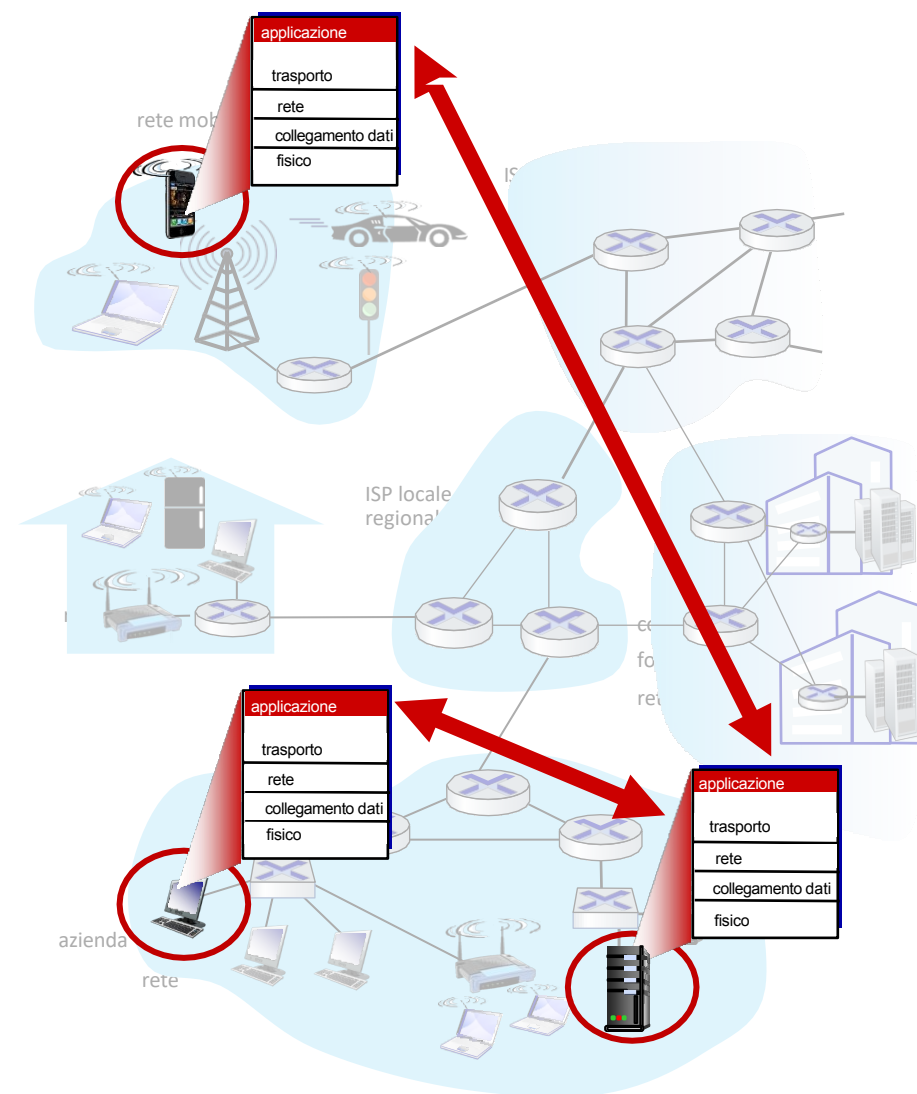
Creazione di un'app di rete

scrivere programmi che:

- funziona su sistemi finali (diversi)
- comunicano tramite rete
- ad esempio, il software del server web comunica con il software del browser

non è necessario scrivere software per i dispositivi di rete principali

- I dispositivi di rete core non eseguono applicazioni utente
- applicazioni sui sistemi finali consente un rapido sviluppo delle applicazioni e la loro diffusione



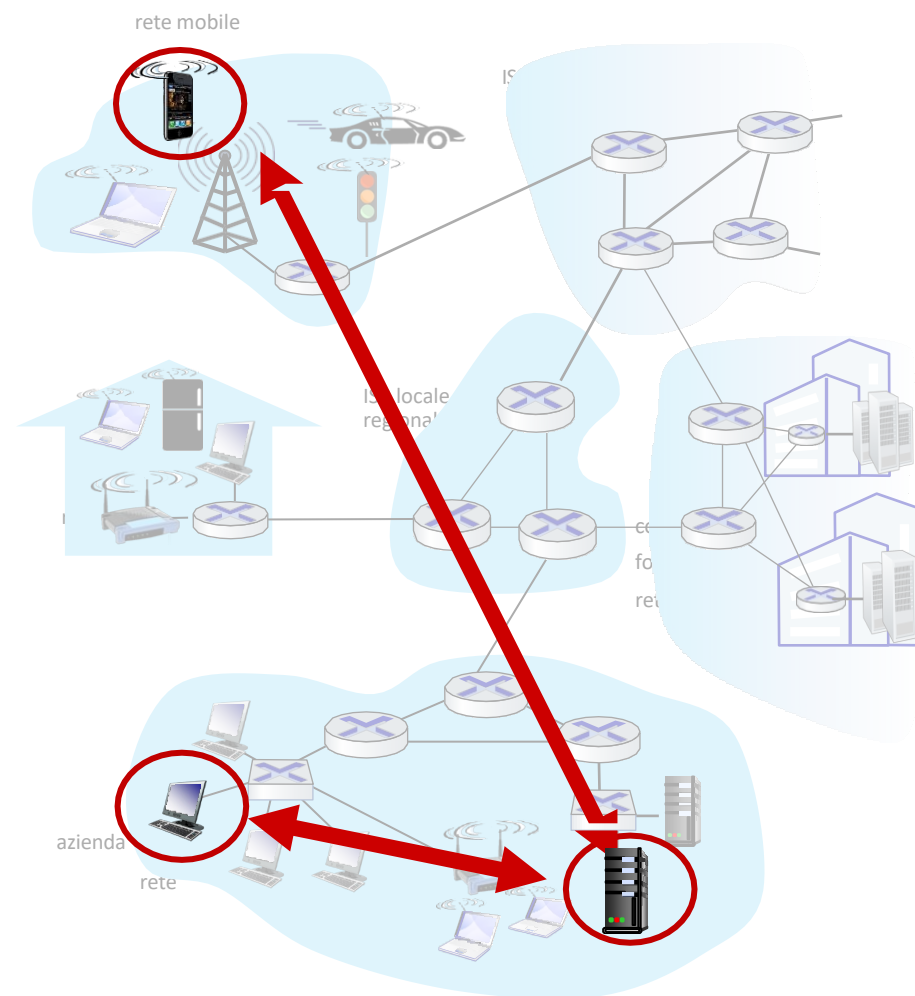
Paradigma client-server

server:

- host sempre attivo
- indirizzo IP permanente
- spesso nei data center, per il ridimensionamento

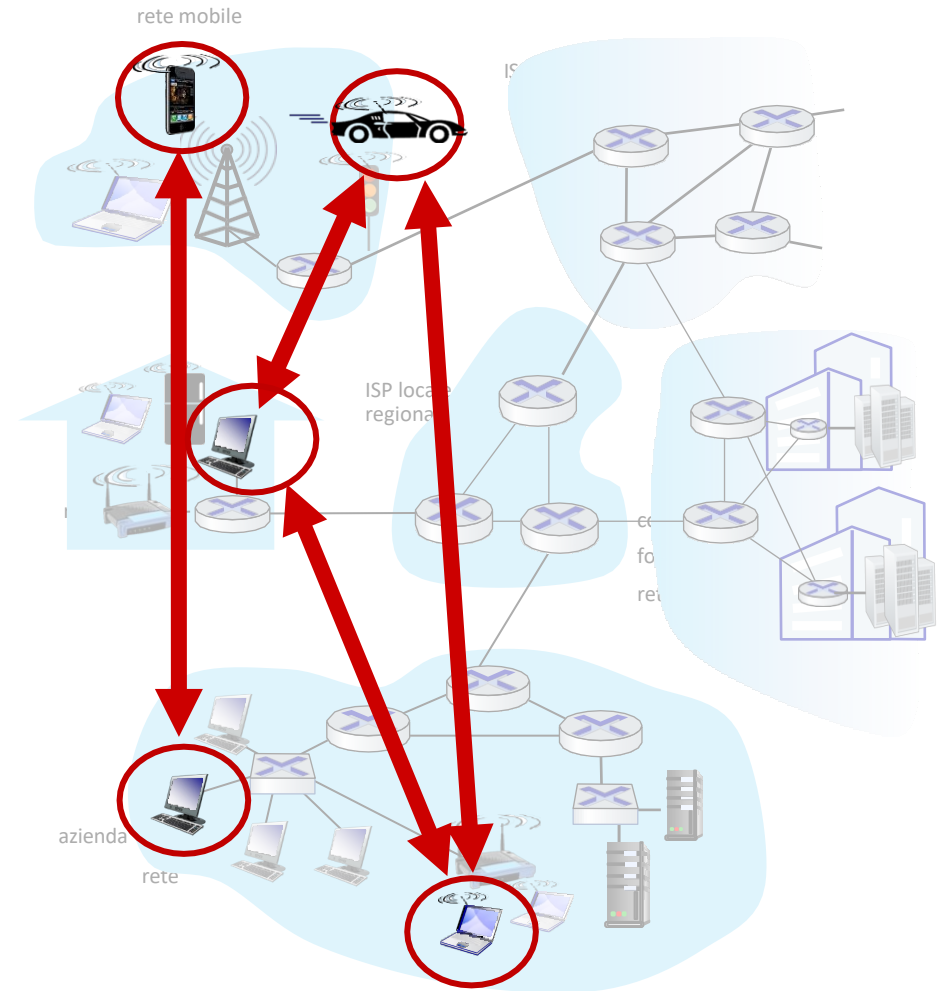
clienti:

- contatto, comunicazione con il server
- possono essere connessi in modo intermittente
- possono avere indirizzi IP dinamici
- *non* comunicano direttamente tra loro
- esempi: HTTP, IMAP, FTP



Architettura peer-to-peer

- *nessun* server sempre attivo
- sistemi finali arbitrari comunicano direttamente
- i peer richiedono servizi ad altri peer e forniscono servizi ad altri peer in cambio
 - *autoscalabilità*: i nuovi peer apportano nuova capacità di servizio, nonché nuove richieste di servizio
- i peer sono connessi in modo intermittente e cambiano indirizzo IP
 - gestione complessa
- esempio: condivisione di file P2P [BitTorrent]



Processi di comunicazione

processo: programma in esecuzione all'interno di un host

- all'interno dello stesso host, due processi comunicano utilizzando **la comunicazione interprocessuale** (definita dal sistema operativo)
- i processi in host diversi comunicano scambiandosi **messaggi**

client, server

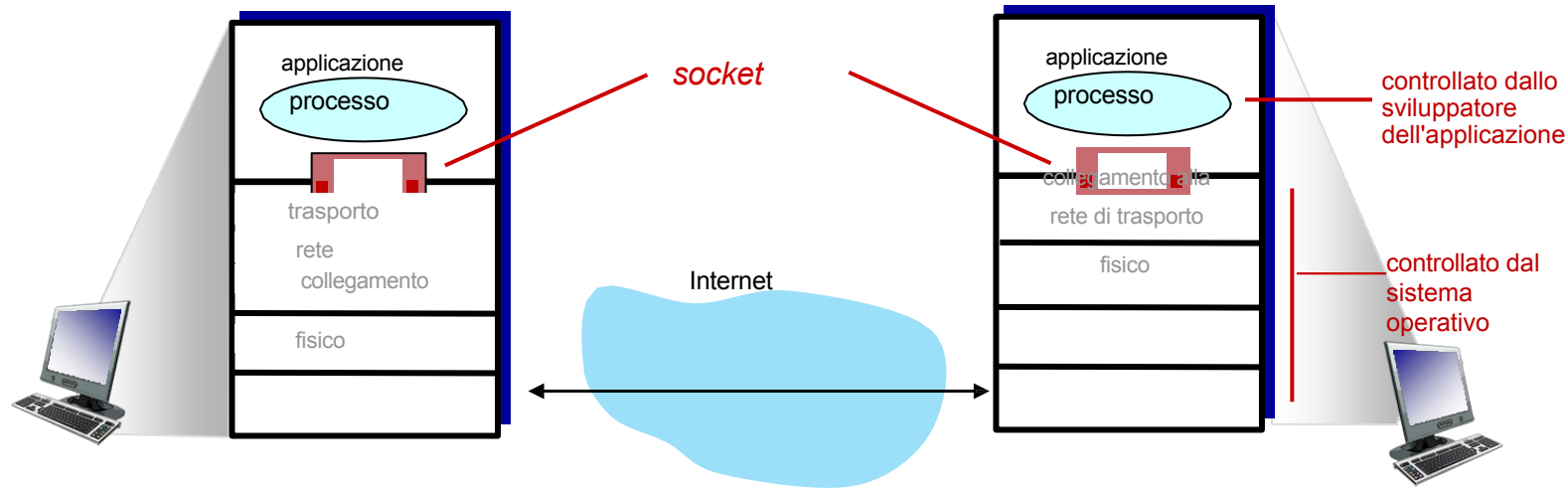
processo client: processo che avvia la comunicazione

processo server: processo che attende di essere contattato

- nota: le applicazioni con architetture P2P hanno processi client e processi server

Socket

- il processo invia/riceve messaggi al/dal proprio **socket**
- il socket è analogo a una porta
 - il processo di invio spinge il messaggio fuori dalla porta
 - il processo di invio si affida all'infrastruttura di trasporto dall'altra parte della porta per consegnare il messaggio al socket del processo di ricezione
 - due socket coinvolti: uno su ciascun lato



Indirizzamento dei processi

- per ricevere messaggi, il processo deve avere *un identificatore*
- il dispositivo host ha un indirizzo IP unico a 32 bit
- D: l'indirizzo IP dell'host su cui viene eseguito il processo è sufficiente per identificare il processo?
 - R: no, *molti* processi possono essere in esecuzione sullo stesso host
- *L'identificatore* include sia l'indirizzo IP che i numeri di porta associati al processo sull'host.
- Esempi di numeri di porta:
 - Server HTTP: 80
 - server di posta: 25
- Per inviare un messaggio HTTP al server web gaia.cs.umass.edu:
 - Indirizzo IP: 128.119.245.12
 - numero di porta: 80
- più brevemente...

Di quale servizio di trasporto ha bisogno un'app?

integrità dei dati

- alcune app (ad esempio, trasferimento di file, transazioni web) richiedono un trasferimento dati affidabile al 100%
- altre app (ad esempio, audio) possono tollerare alcune perdite

tempistica

- alcune app (ad esempio, telefonia Internet, giochi interattivi) richiedono un ritardo basso per essere "efficaci"

velocità

- alcune app (ad es. multimedia) richiedono una quantità minima di throughput per essere "efficaci"
- altre applicazioni ("applicazioni elastiche") utilizzano qualsiasi throughput ottengano

sicurezza

- crittografia, integrità dei dati,
...

Requisiti del servizio di trasporto: app comuni

applicazione	perdita di dati	throughput	sensibile al tempo?
trasferimento/download di file	nessuna	elastico	no
e-mail	perdita	elastico	no
Documenti web	nessuna	elastico	no
audio/video in tempo reale	perdita	audio: 5 Kbps-1 Mbps video: 10 Kbps-5 Mbps come sopra Kbps+	sì, 10 millisecondi
streaming audio/video giochi	nessuna	elastico	sì, pochi secondi sì, 10 millisecondi sì e no
interattivi messaggistica di testo	perdita		
	tollerante alla perdita		
	tollerante alla perdita		
	nessuna perdita		

Servizi di protocolli di trasporto Internet

Servizio TCP:

- *trasporto affidabile* tra il processo di invio e quello di ricezione
- *controllo di flusso*: il mittente non sovraccarica il destinatario
- *controllo della congestione*: limitazione del mittente in caso di sovraccarico della rete
- *orientato alla connessione*: richiede la configurazione tra i processi client e server
- *non fornisce*: temporizzazione, garanzia di throughput minimo, sicurezza

Servizio UDP:

- *trasferimento dati inaffidabile* tra il processo di invio e quello di ricezione
- *non fornisce*: affidabilità, controllo di flusso, controllo della congestione, temporizzazione, garanzia di throughput, sicurezza o configurazione della connessione.

D: perché preoccuparsi? *Perché*
esiste un UDP?

Applicazioni Internet e protocolli di trasporto

applicazione		protocollo layer	protocollo di trasporto
trasferimento/download di file		FTP [RFC 959]	TCP
			TCP
e-mail		SMTP [RFC 5321]	TCP
Documenti web		HTTP [RFC 7230, 9110]	TCP o UDP
Telefonia		SIP [RFC 3261],	
Internet		RTP [RFC	
		3550] o HTTP proprietario [RFC 7230],	
		DASH	TCP
streaming audio/video		WOW, FPS (proprietario)	UDP o TCP
giochi interattivi			

Protezione TCP

Socket TCP e UDP standard:

- nessuna crittografia
- le password in chiaro inviate al socket attraversano Internet in chiaro (!)

Transport Layer Security (TLS)

- fornisce connessioni TCP crittografate
- integrità dei dati
- autenticazione degli endpoint

TLS implementato nel livello applicativo

- le app utilizzano librerie TLS, che a loro volta utilizzano TCP
- testo in chiaro inviato al "socket" attraversa Internet *crittografato*
- ulteriori informazioni: Capitolo 8

Livello applicativo: panoramica

- Principi delle applicazioni di rete
- **Web e HTTP**
- E-mail, SMTP, IMAP
- Il sistema dei nomi di dominio DNS
- Applicazioni P2P
- Streaming video e reti di distribuzione dei contenuti
- Programmazione socket con UDP e TCP



Web e HTTP

Innanzitutto, una breve panoramica...

- una pagina web è composta da *oggetti*, ciascuno dei quali può essere memorizzato su diversi server web
- un oggetto può essere un file HTML, un'immagine JPEG, un'applet Java, un file audio...
- una pagina web è costituita da *un file HTML di base* che include *diversi oggetti referenziati, ciascuno* indirizzabile tramite un *URL*, ad esempio

`www.someschool.edu/someDept/pic.gif`

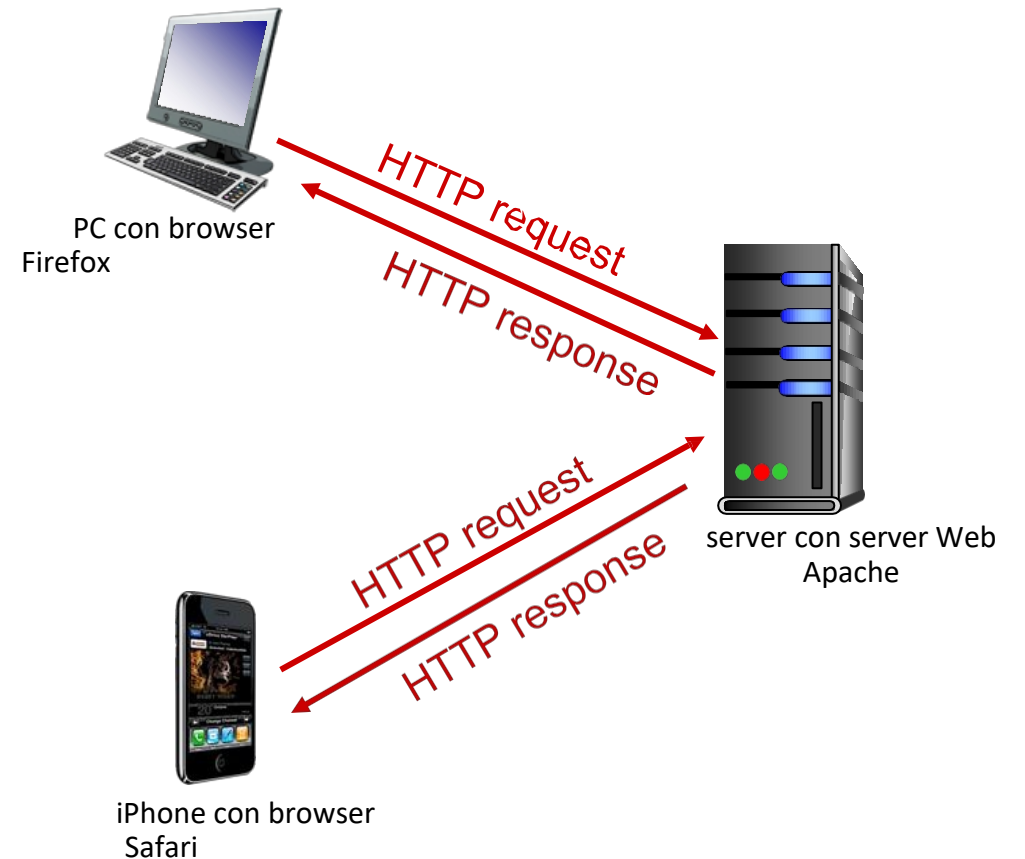
hostname

nome del percorso

Panoramica HTTP

HTTP: protocollo di trasferimento ipertestuale

- Protocollo del livello applicativo del Web
- Modello client/server:
 - *client*: browser che richiede, riceve (utilizzando il protocollo HTTP) e "visualizza" oggetti Web
 - *server*: il server Web invia (utilizzando il protocollo HTTP) oggetti in risposta alle richieste



Panoramica HTTP (continua)

HTTP utilizza TCP:

- il client avvia una connessione TCP (crea un socket) al server, porta 80
- il server accetta la connessione TCP dal client
- Messaggi HTTP (messaggi del protocollo a livello di applicazione) scambiati tra il browser (client HTTP) e il server Web (server HTTP)
- Connessione TCP chiusa

HTTP è "stateless"

- il server *non* conserva *alcuna* informazione sulle richieste precedenti del client

protocolli che mantengono lo "stato"
sono complessi!

- è necessario conservare la cronologia (stato) passata
- se il server/client si blocca, il loro le opinioni sullo "stato" possono essere incoerenti, devono essere riconciliate

Connessioni HTTP: due tipi

HTTP non persistente

1. Connessione TCP aperta
2. al massimo un oggetto inviato tramite connessione TCP
3. Connessione TCP chiusa

il download di più oggetti richiede più connessioni

HTTP persistente

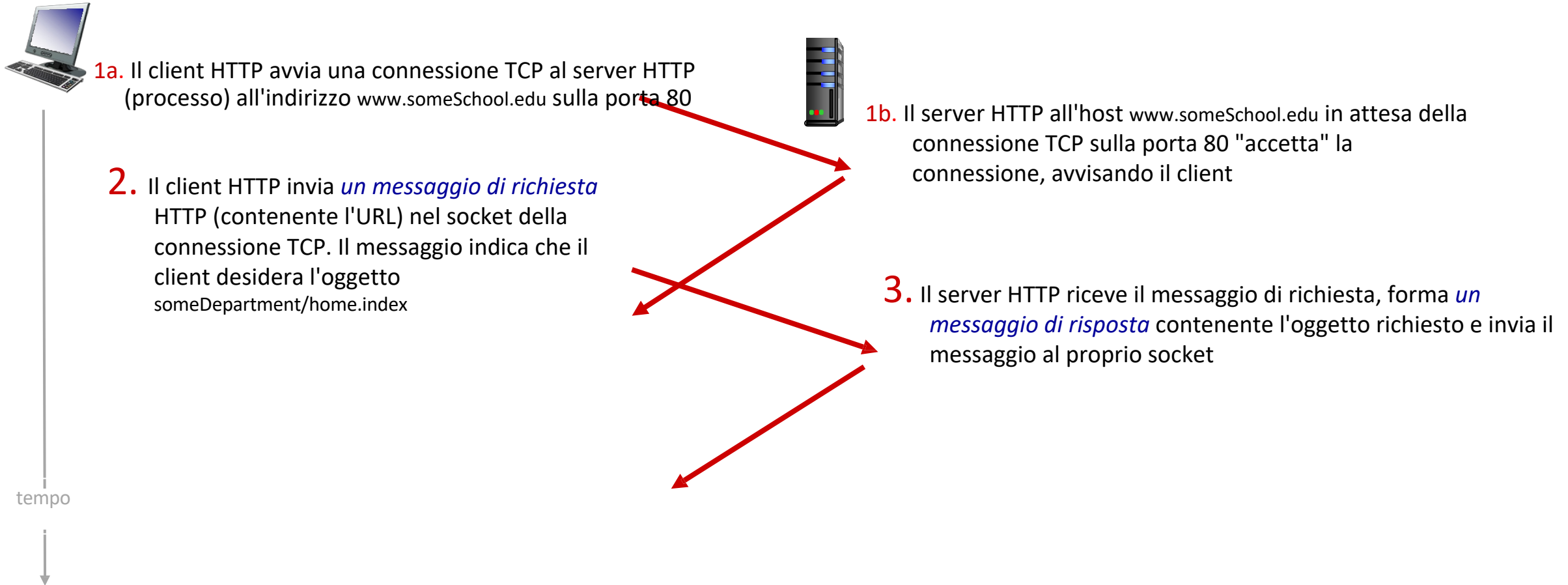
- Connessione TCP aperta a un server
- È possibile inviare più oggetti tramite una *singola* connessione TCP tra il client e quel server
- Connessione TCP chiusa

HTTP non persistente: esempio

L'utente inserisce l'URL:

`www.someSchool.edu/someDepartment/home.index`

(contenente testo, riferimenti a 10 immagini jpeg)



HTTP non persistente: esempio (continua)

L'utente inserisce l'URL:

`www.someSchool.edu/someDepartment/home.index`

(contenente testo, riferimenti a 10 immagini jpeg)



4. Il server HTTP chiude la connessione TCP.

5. Il client HTTP riceve il messaggio di risposta contenente il file html e visualizza l'html. Analizzando il file html, trova 10 oggetti jpeg referenziati

6. I passaggi da 1 a 5 vengono ripetuti per ciascuno dei 10 oggetti jpeg

tempo



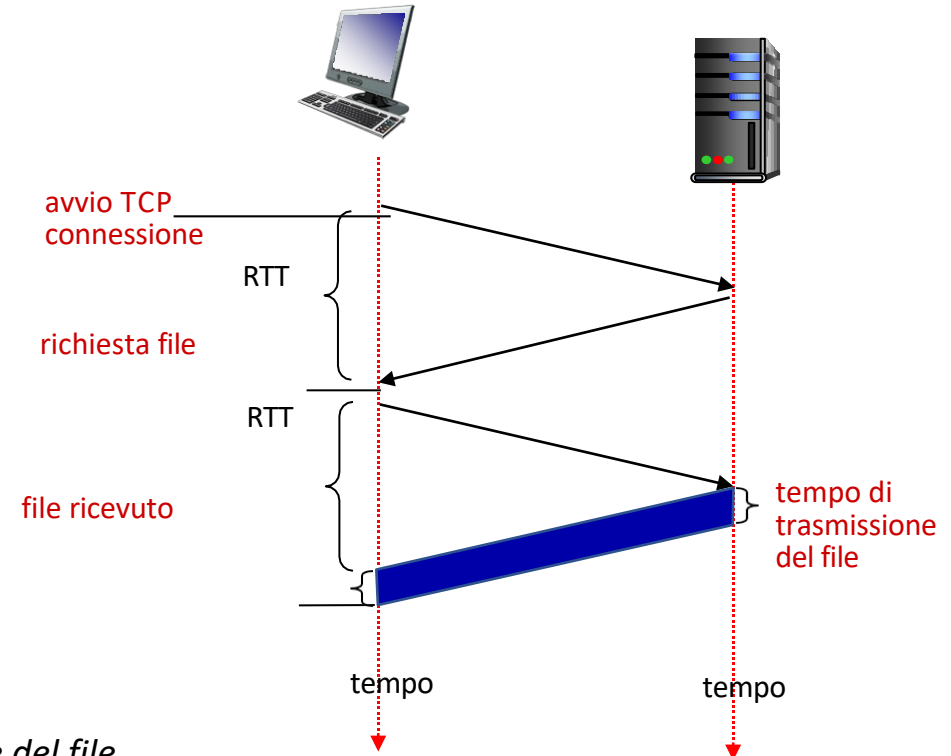
HTTP non persistente: tempo di risposta

RTT (definizione): tempo impiegato da un piccolo pacchetto per viaggiare dal client al server e viceversa

Tempo di risposta HTTP (per oggetto):

- un RTT per avviare la connessione TCP
- un RTT per la richiesta HTTP e i primi byte della risposta HTTP da restituire
- tempo di trasmissione dell'oggetto/file

Tempo di risposta HTTP non persistente = $2RTT + \text{tempo di trasmissione del file}$



HTTP persistente (HTTP 1.1)

Problemi HTTP non persistente:

- richiede 2 RTT per oggetto
- Overhead del sistema operativo per *ogni* connessione TCP
- i browser spesso aprono più connessioni TCP parallele per recuperare gli oggetti referenziati in parallelo

HTTP persistente (HTTP1.1):

- Il server lascia aperta la connessione dopo aver inviato la risposta
- i messaggi HTTP successivi tra lo stesso client/server vengono inviati tramite la connessione aperta
- il client invia richieste non appena incontra un oggetto referenziato
- solo un RTT per tutti gli oggetti referenziati (riducendo il tempo di risposta della metà)

Messaggio di richiesta HTTP

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

riga di richiesta (GET, POST,
comandi HEAD)

intestazione
righe

ritorno a capo, avanzamento riga all'inizio
della riga indica

fine delle righe dell'intestazione

GET /index.html HTTP/1.1\r\n

Host: www-net.cs.umass.edu\r\n

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101
Firefox/80.0 \r\n

Accetta: text/html,application/xhtml+xml\r\n Accetta lingua: en-
us,en;q=0.5\r\n

Accetta codifica: gzip, deflate\r\n

Connessione: keep-alive\r\n

\r\n

carattere di ritorno a capo

carattere di avanzamento riga

* Per ulteriori informazioni, consulta gli esercizi interattivi online

esempi: http://gaia.cs.umass.edu/kurose_ross/interactive/

Altri messaggi di richiesta HTTP

Metodo POST:

- le pagine web spesso includono moduli di inserimento dati
- dati inseriti dall'utente inviati dal client al server nel corpo dell'entità del messaggio di richiesta HTTP POST

Metodo GET (per l'invio di dati al server):

- includere i dati dell'utente nel campo URL dell'HTTP messaggio di richiesta GET (dopo un "?"):

`www.somesite.com/animalsearch?monkeys&banana`

Metodo HEAD:

- richieste di intestazioni (solo) che verrebbero restituite se l'URL specificato fosse richiesto con un metodo HTTP GET.

Metodo PUT:

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

Messaggio di risposta HTTP

riga di stato (codice di stato del protocollo,
frase di stato)

HTTP/1.1 200 OK

Data: martedì, 08 settembre 2020 00:53:20 GMT

Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips

PHP/7.4.9 mod_perl/2.0.11 Perl/v5.16.3

Ultima modifica: martedì 1 marzo 2016 18:57:50 GMT ETag: "a5b-52d015789ee9e"

Accetta intervalli: byte

Lunghezza contenuto: 2651

Tipo di contenuto: text/html; charset=UTF-8

\r\n

dati dati dati dati dati ...

intestazione

righe

dati, ad esempio file HTML richiesto

* Per ulteriori esempi, consulta gli esercizi interattivi online: http://gaia.cs.umass.edu/kurose_ross/interactive/

Codici di stato delle risposte HTTP

- Il codice di stato appare nella prima riga del messaggio di risposta dal server al client.
- Alcuni codici di esempio: **200 OK**
 - richiesta riuscita, oggetto richiesto più avanti in questo messaggio**301 Spostato in modo permanente**
 - oggetto richiesto spostato, nuova posizione specificata più avanti in questo messaggio (nel campo Location:)**400 Richiesta errata**
 - messaggio di richiesta non compreso dal server**404 Non trovato**
 - documento richiesto non trovato su questo server**505 Versione HTTP non supportata**

Prova HTTP (lato client) per conto tuo

1. netcat sul tuo server Web preferito:

```
% nc -c -v gaia.cs.umass.edu 80
```

- apre una connessione TCP alla porta 80 (porta predefinita del server HTTP) su gaia.cs.umass.edu.
- Tutto ciò che viene digitato verrà inviato alla porta 80 su gaia.cs.umass.edu

2. digita una richiesta HTTP GET:

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

- digitando questo (premere due volte il tasto Invio), si invia questa richiesta GET minima (ma completa) al server HTTP server

3. guarda il messaggio di risposta inviato dal server HTTP!

(oppure usa Wireshark per vedere la richiesta/risposta HTTP catturata)

Mantenimento dello stato utente/server: cookie

I siti web e i browser dei clienti utilizzano *i cookie* per mantenere uno stato tra le transazioni

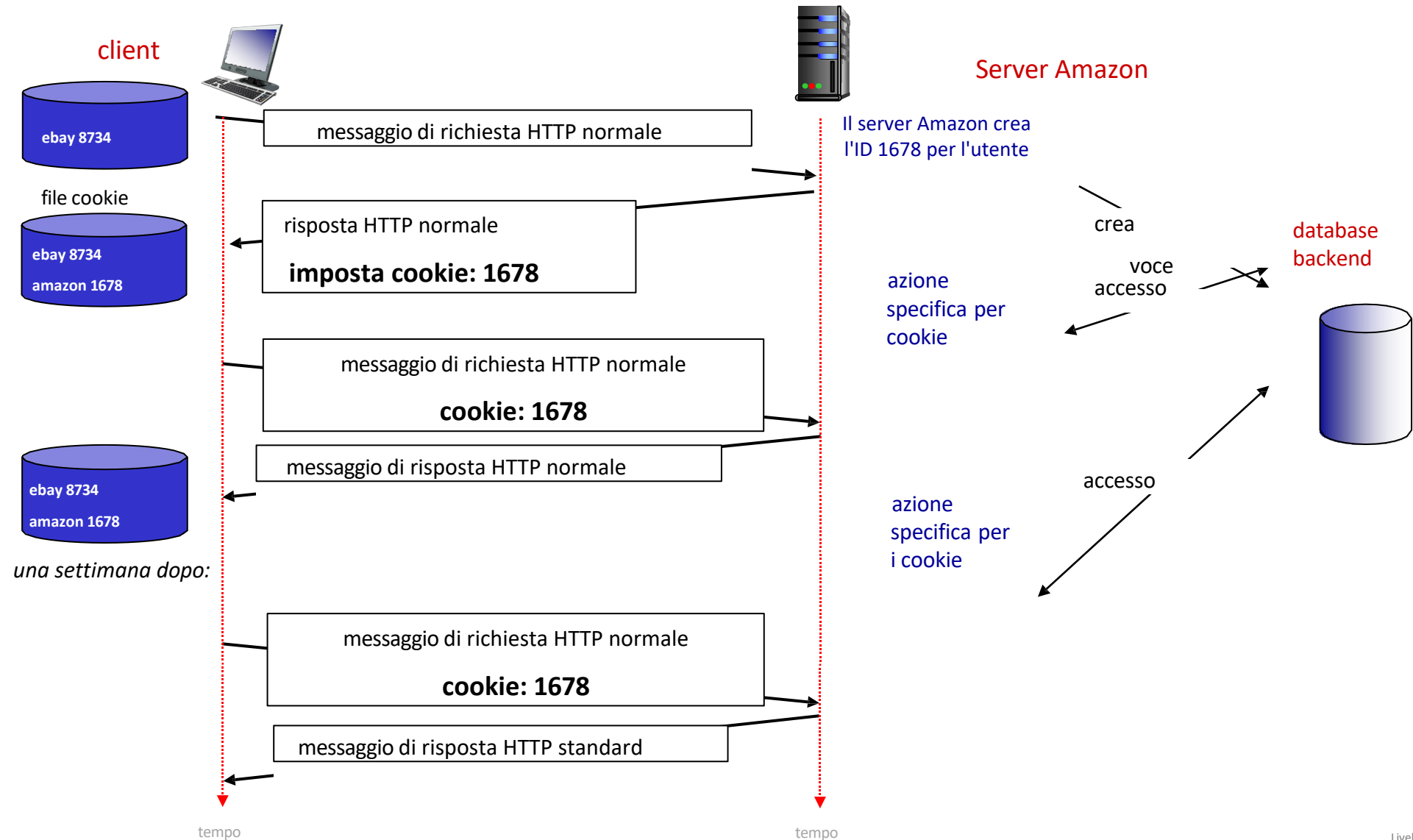
quattro componenti:

- 1) intestazione del cookie linea di *risposta* HTTP
messaggio
- 2) riga dell'intestazione del cookie nel successivo messaggio
HTTP
messaggio *di richiesta*
- 3) file cookie conservato sull'host dell'utente, gestito dal browser dell'utente
- 4) database back-end sul sito Web

Esempio:

- Susan utilizza il browser sul proprio laptop e visita per la prima volta un determinato sito di e-commerce
- quando le richieste HTTP iniziali arrivano al sito, il sito crea:
 - ID univoco (noto anche come "cookie")
 - voce nel database back-end per l'ID
- le successive richieste HTTP di Susan a questo sito conterranno il valore dell'ID del cookie, consentendo al sito di "identificare" Susan

Mantenimento dello stato utente/server: cookie



Cookie HTTP: commenti

A cosa servono i cookie:

- autorizzazione
- carrelli della spesa
- raccomandazioni
- stato della sessione utente (e-mail web)

Sfida: come mantenere lo stato?

- *agli endpoint del protocollo:* mantenere lo stato presso il mittente/destinatario su più transazioni
- *nei messaggi:* i cookie nei messaggi HTTP trasportano lo stato

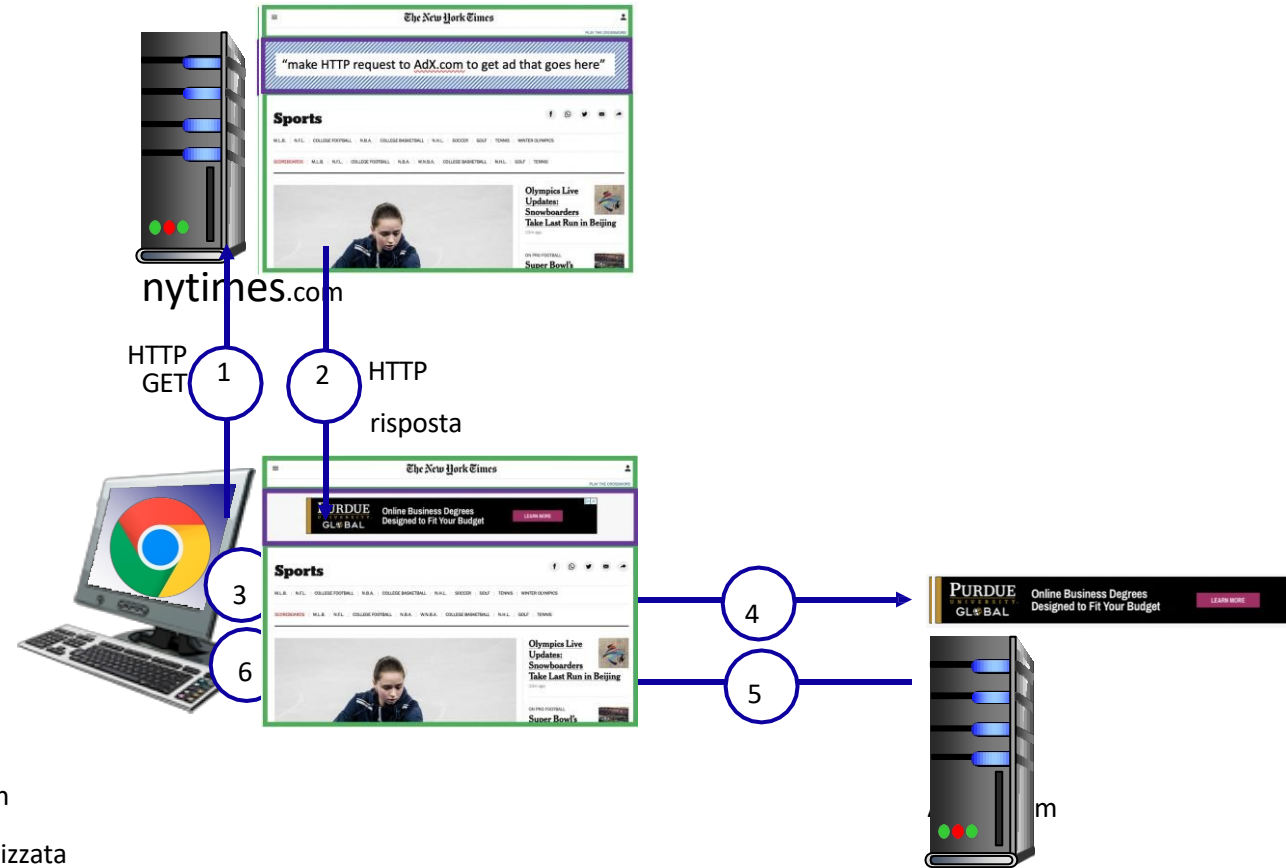
a parte

cookie e privacy:

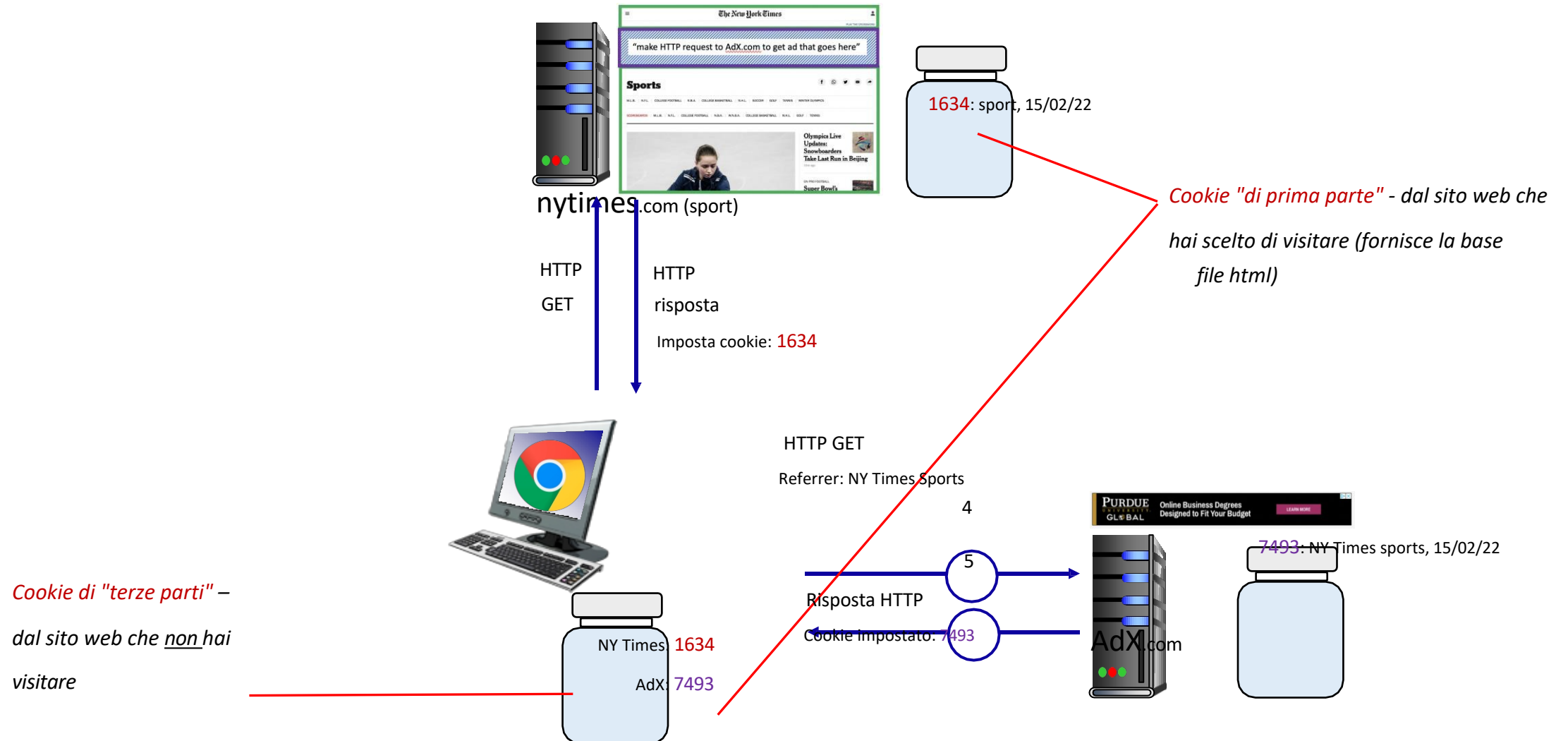
- i cookie consentono ai siti di *raccogliere* molte informazioni su di te quando visiti il loro sito.
- I cookie persistenti di terze parti (cookie di tracciamento) consentono di tracciare l'identità comune (valore del cookie) su più siti web

Esempio: visualizzazione di una pagina web del NY Times

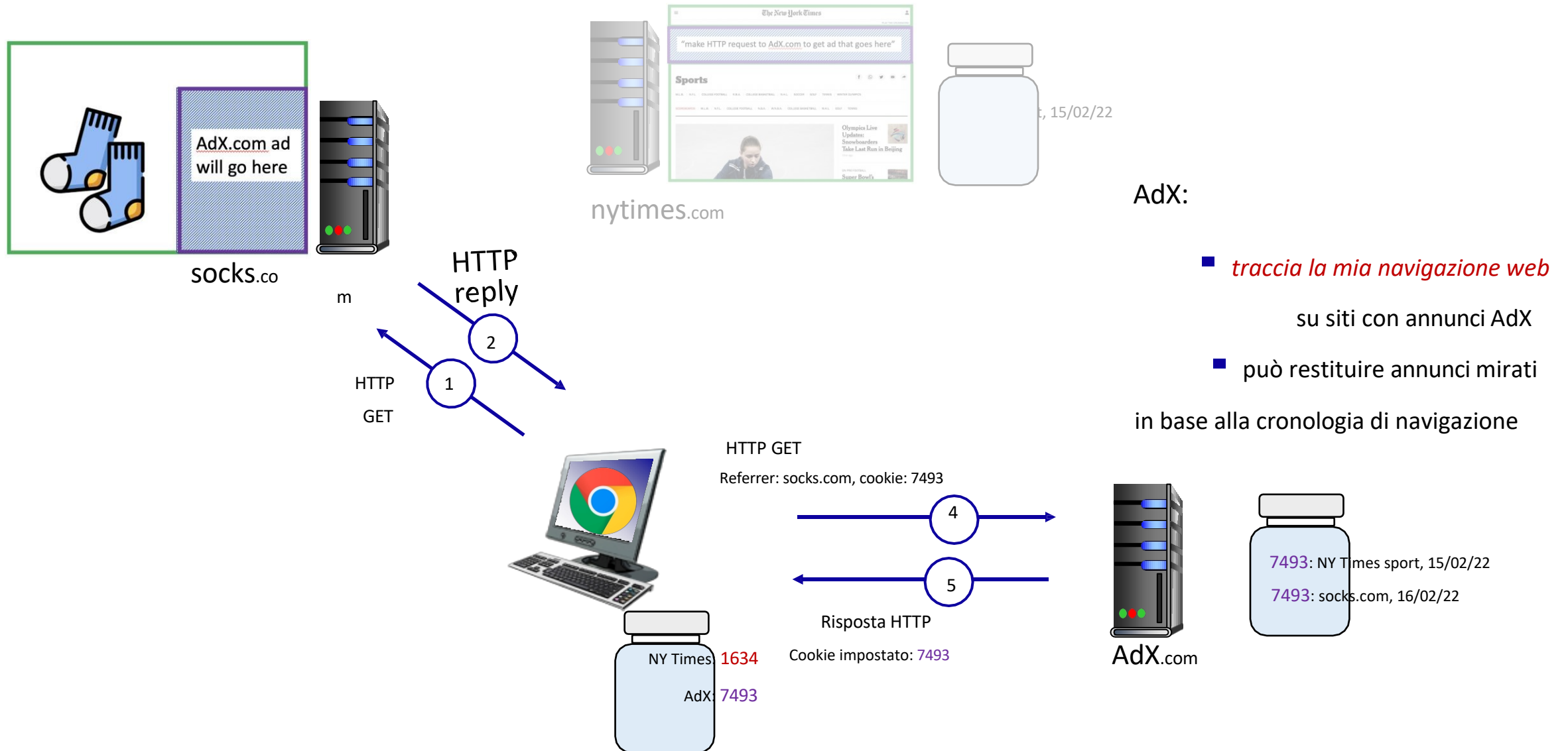
- 1 GET file html di base da nytimes.com
- 2
- 4 recupera l'annuncio da AdX.com
- 5
- 7 visualizza pagina composta



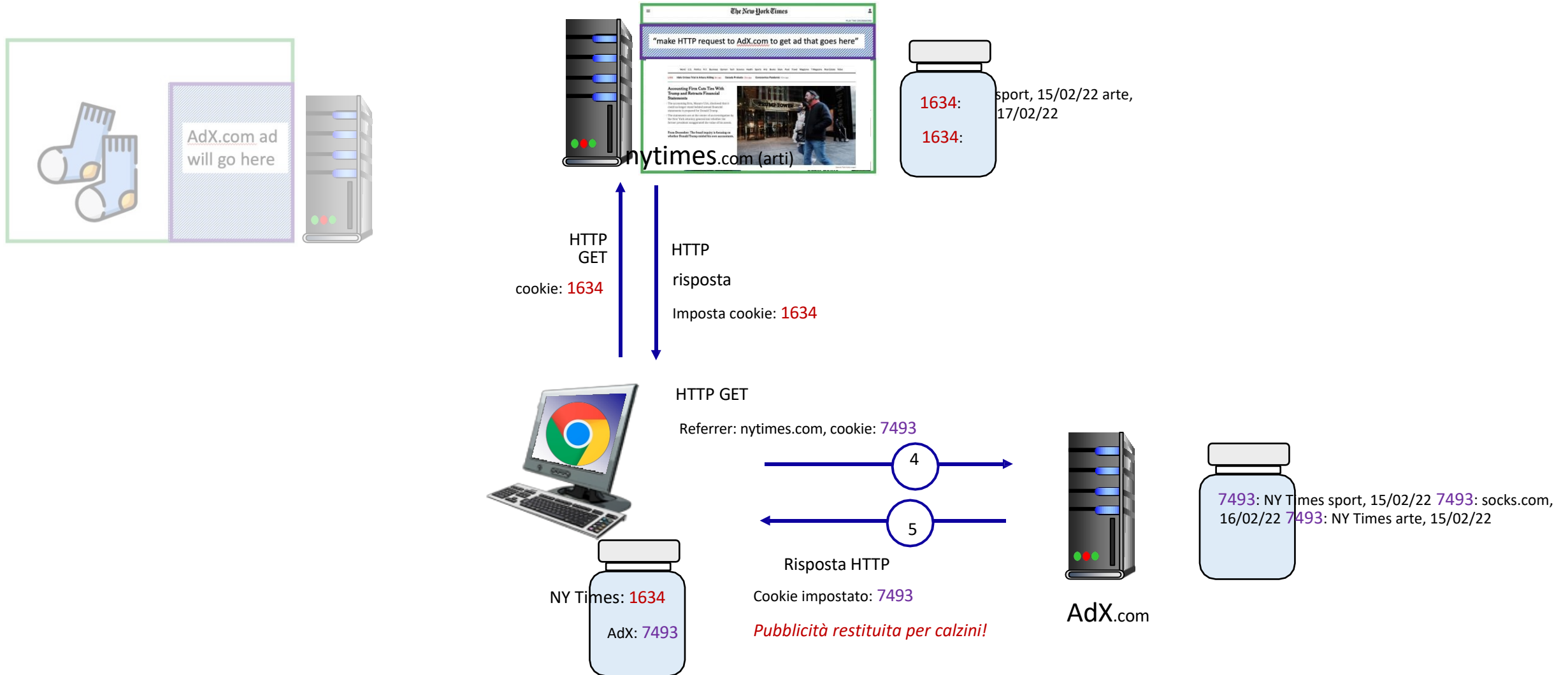
Cookie: tracciamento del comportamento di navigazione dell'utente



Cookie: tracciamento del comportamento di navigazione dell'utente



Cookie: tracciamento del comportamento di navigazione di un utente (un giorno dopo)



Cookie: tracciamento del comportamento di navigazione di un utente

I cookie possono essere utilizzati per:

- tracciare il comportamento dell'utente su un determinato sito web (**cookie di prima parte**)
- tracciare il comportamento dell'utente su più siti web (**cookie di terze parti**) senza che l'utente abbia mai scelto di visitare il sito di tracciamento (!)
- il tracciamento può essere *invisibile* all'utente:
 - anziché visualizzare un annuncio pubblicitario che attiva HTTP GET al tracker, potrebbe essere un link invisibile

tracciamento di terze parti tramite cookie:

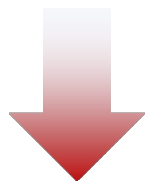
- disabilitato di default nei browser Firefox e Safari
- da disabilitare nel browser Chrome nel 2023

GDPR (Regolamento generale sulla protezione dei dati dell'UE) e cookie

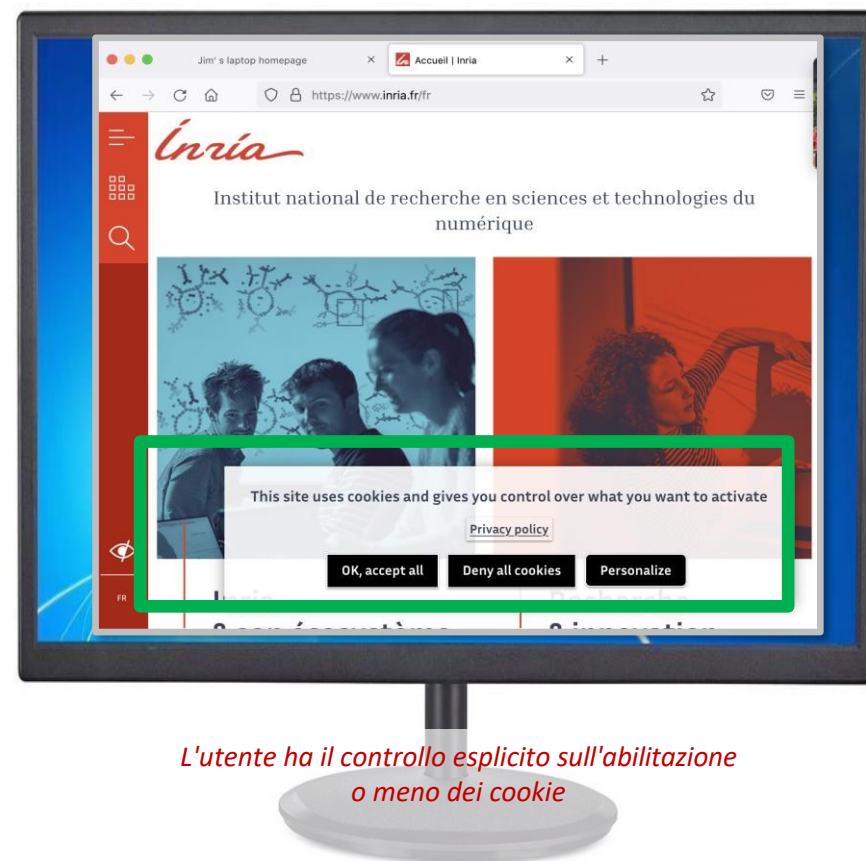
"Le persone fisiche possono essere associate a identificatori online [...] quali indirizzi IP, identificatori cookie o altri identificatori [...].

Ciò può lasciare tracce che, in particolare se combinate con identificatori univoci e altre informazioni ricevute dai server, possono essere utilizzate per creare profili delle persone fisiche e identificarle".

GDPR, considerando 30 (maggio 2018)



quando i cookie possono identificare un individuo, i cookie sono considerati dati personali, soggetti alle norme sul trattamento dei dati personali del GDPR

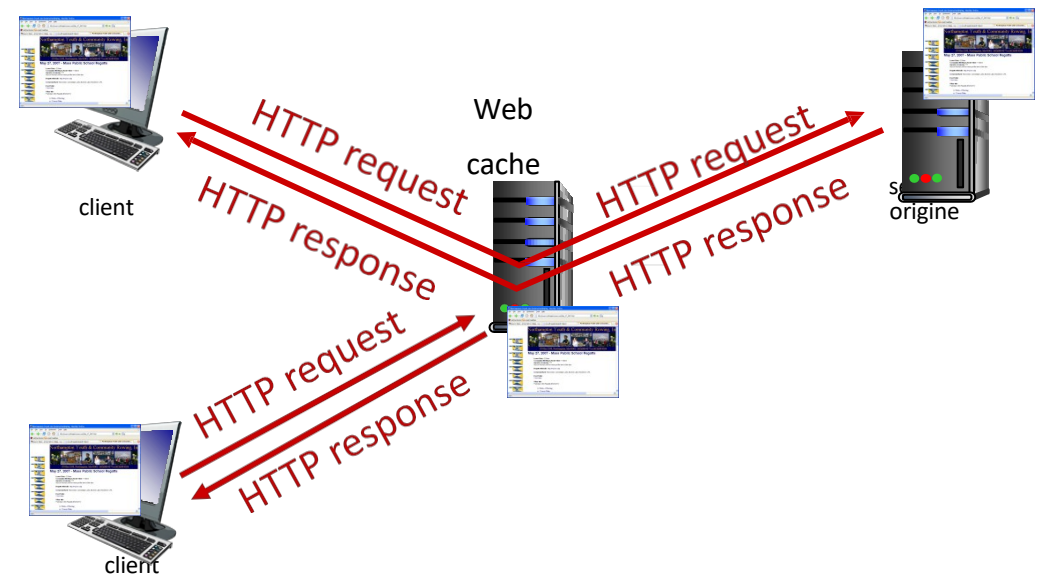


L'utente ha il controllo esplicito sull'abilitazione o meno dei cookie

Cache web

Obiettivo: soddisfare le richieste dei clienti senza coinvolgere il server di origine

- l'utente configura il browser in modo che punti a una **cache web** (locale)
- browser invia tutte le richieste HTTP alla cache
 - *se* l'oggetto è nella cache: la cache restituisce l'oggetto al client
 - *altrimenti* la cache richiede l'oggetto al server di origine, memorizza l'oggetto ricevuto nella cache, quindi restituisce l'oggetto al client



Cache web (ovvero server proxy)

- La cache web funge sia da client che da server
 - server per il client che ha effettuato la richiesta originale
 - client verso il server di origine
- il server comunica alla cache la cache consentita dell'oggetto nell'intestazione della risposta:

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

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

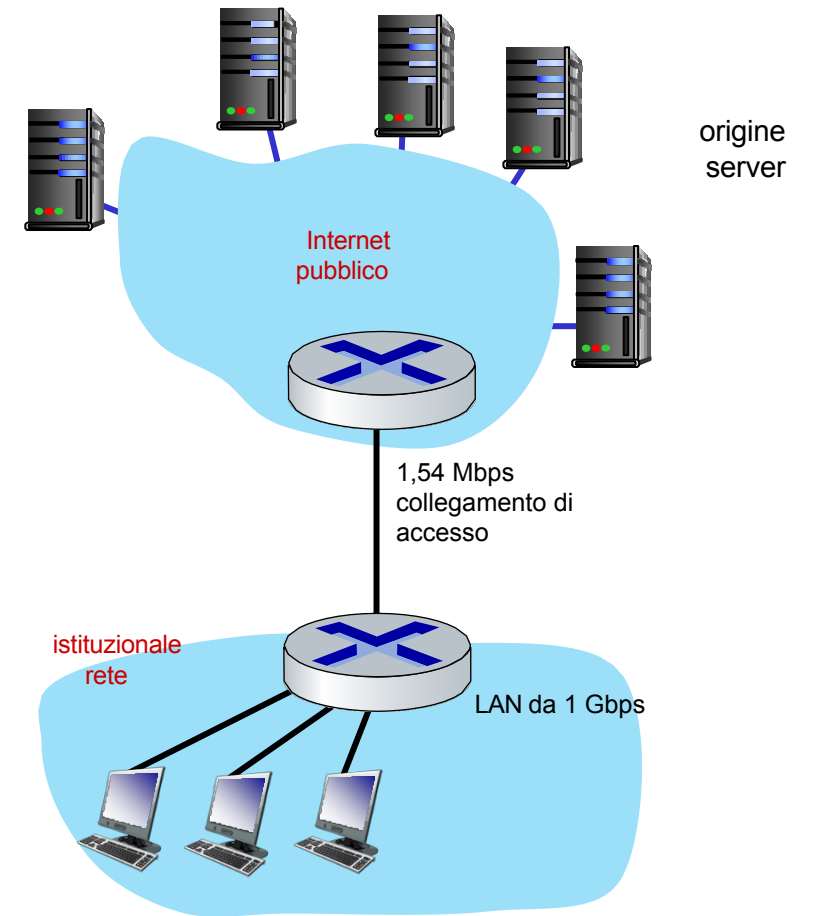
Perché il caching web?

- ridurre il tempo di risposta per la richiesta del client
 - La cache è più vicina al client
- ridurre il traffico sul collegamento di accesso di un'istituzione
- Internet è denso di cache
 - Consente ai fornitori di contenuti "poveri" a fornire contenuti in modo più efficace

Esempio di cache

Scenario:

- velocità di accesso: 1,54 Mbps
- RTT dal router istituzionale al server: 2 sec
- dimensione oggetto web: 100K bit
- frequenza media delle richieste dai browser ai server di origine: 15/sec
 - velocità media di trasmissione dati ai browser: 1,50 Mbps



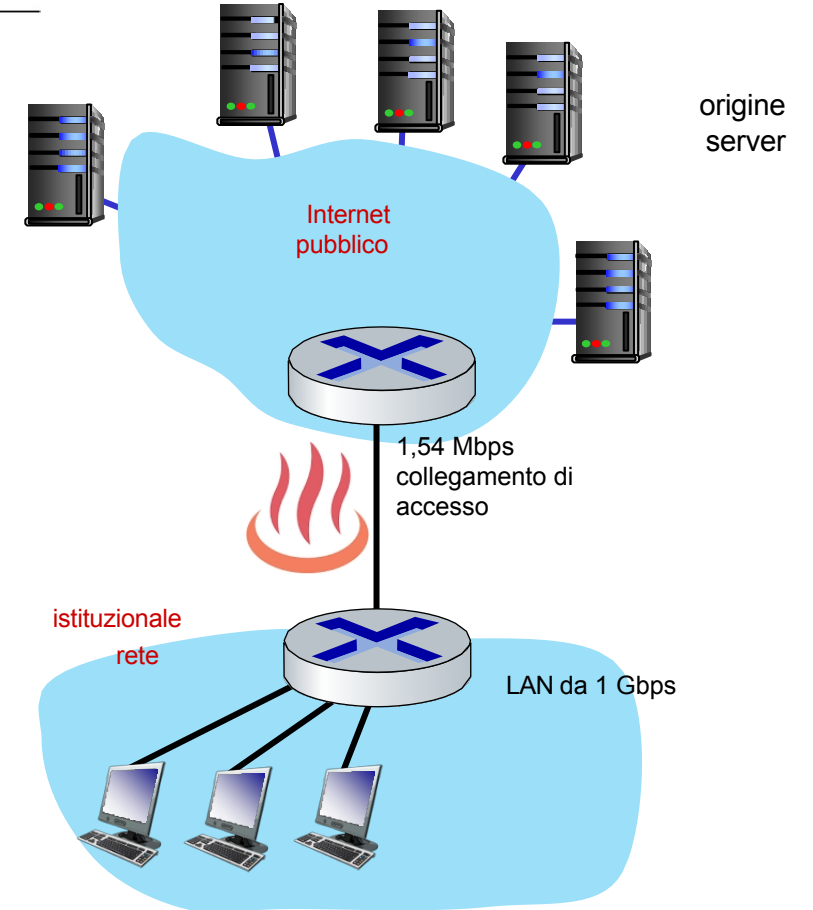
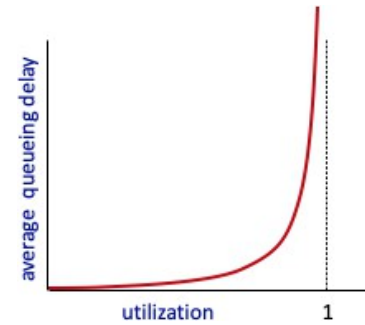
Esempio di caching

Scenario:

- velocità di accesso: 1,54 Mbps
- RTT dal router istituzionale al server: 2 sec
- dimensione oggetto web: 100K bit
- velocità media di richiesta dai browser ai server di origine: 15/sec
 - velocità media di trasmissione dati ai browser: 1,50 Mbps

Prestazioni:

- utilizzo del collegamento di accesso = 0,97
- Utilizzo LAN: .0015
- ritardo end-to-end = ritardo Internet +
ritardo del collegamento di accesso + ritardo
LAN
= 2 sec + minuti + microsecondi



Opzione 1: acquistare un collegamento di accesso più veloce

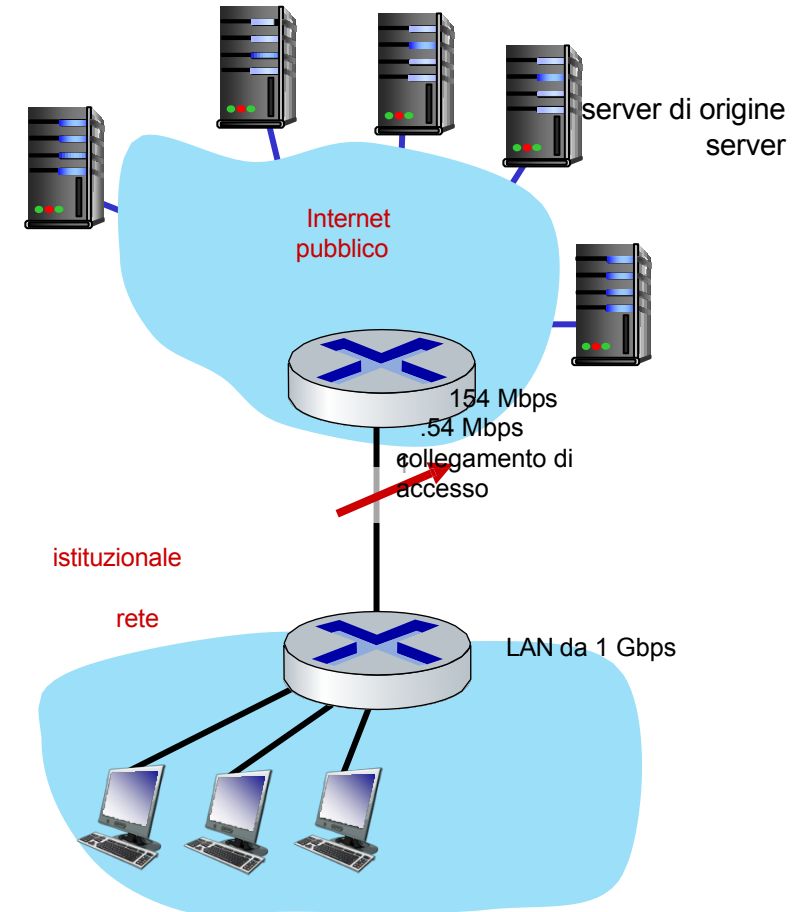
Scenario:

- velocità del collegamento di accesso: 1,54 Mbps
- RTT dal router istituzionale al server: 2 sec
- dimensione dell'oggetto web: 100K bit
- velocità media di richiesta dai browser ai server di origine: 15/sec
 - velocità media di trasmissione dati ai browser: 1,50 Mbps

Prestazioni:

- utilizzo del collegamento di accesso = $\frac{100000}{150000000} = .000667$
- Utilizzo LAN: .0015
- ritardo end-to-end = ritardo Internet +
ritardo del collegamento di accesso + ritardo LAN
= 2 sec + minuti + microsecondi

Costo: collegamento di accesso più veloce (costoso!) msec



Opzione 2: installare una cache web

Scenario:

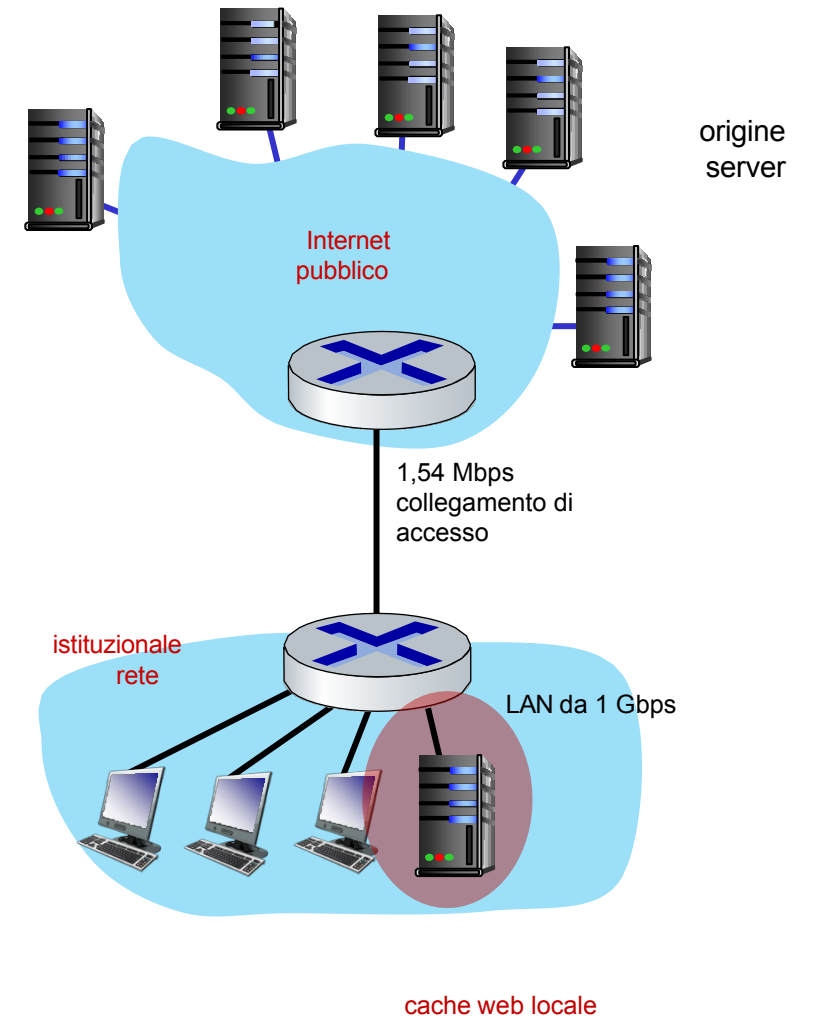
- velocità del collegamento di accesso: 1,54 Mbps
- RTT dal router istituzionale al server: 2 sec
- dimensione oggetto web: 100K bit
- velocità media di richiesta dai browser ai server di origine: 15/sec
 - Velocità media di trasmissione dati ai browser: 1,50 Mbps

Costo: cache web (economico!)

Prestazioni:

- Utilizzo LAN: .?
- utilizzo del collegamento di accesso = ?
- ritardo medio end-to-end = ?

*Come calcolare l'utilizzo del
collegamento
utilizzo, ritardo?*

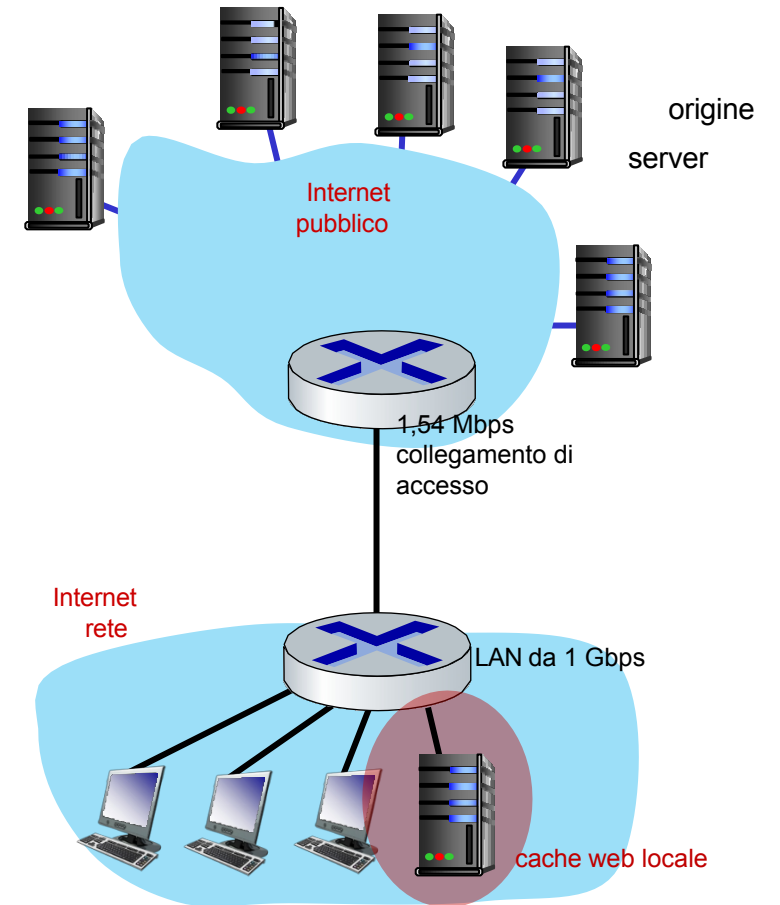


Calcolo dell'utilizzo del collegamento di accesso, ritardo end-to-end

con cache:

supponendo che il tasso di cache hit sia 0,4:

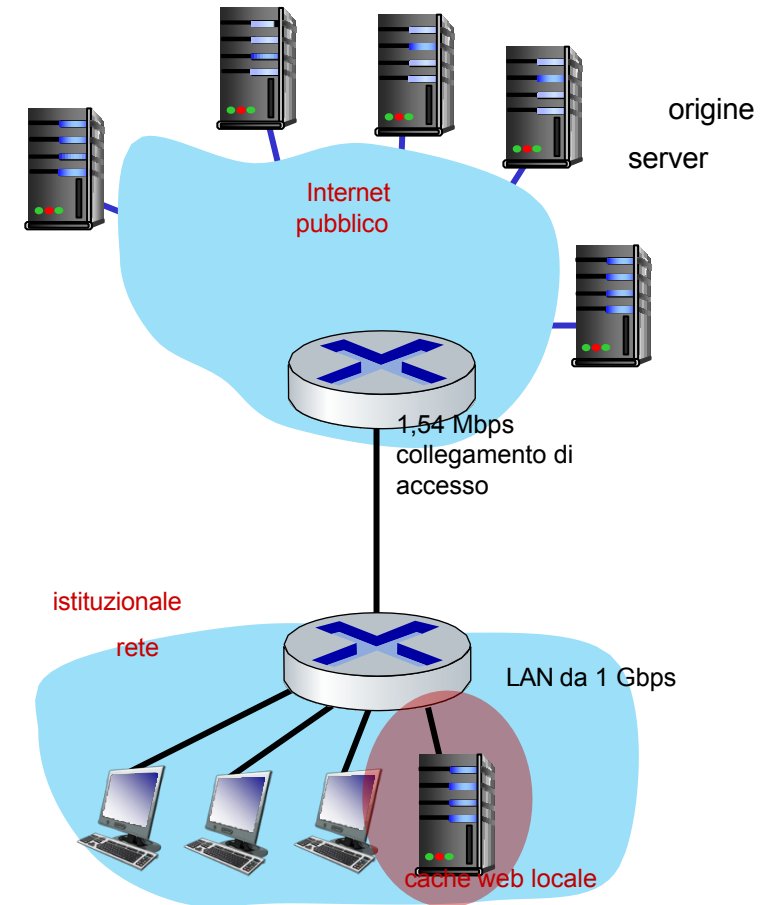
- il 40% delle richieste viene gestito dalla cache, con un basso (msec)
- il 60% delle richieste soddisfatte all'origine
 - velocità ai browser tramite collegamento di accesso
 $= 0,6 * 1,50 \text{ Mbps} = 0,9 \text{ Mbps}$
 - utilizzo del collegamento di accesso $= 0,9/1,54 = 0,58$ significa basso (msec) ritardo di accodamento al collegamento di accesso



Calcolo dell'utilizzo del collegamento di accesso, ritardo end-to-end con cache:

supponendo che il tasso di cache hit sia 0,4:

- il 40% delle richieste viene gestito dalla cache, con un basso (msec)
- il 60% delle richieste soddisfatte all'origine
 - tasso ai browser tramite collegamento di accesso
 $= 0,6 * 1,50 \text{ Mbps} = 0,9 \text{ Mbps}$
 - utilizzo del collegamento di accesso $= 0,9 / 1,54 = 0,58$ significa basso (msec) ritardo di accodamento al collegamento di accesso
- ritardo medio end-to-end:
 $= 0,6 * (\text{ritardo dai server di origine})$
 $+ 0,4 * (\text{ritardo quando soddisfatto nella cache})$
 $= 0,6 (2,01) + 0,4 (\sim \text{msec}) = \sim 1,2 \text{ sec}$



ritardo medio end-to-end inferiore rispetto al collegamento a 154 Mbps (e anche più economico!)

Cache del browser: GET condizionale

Obiettivo: non inviare l'oggetto se il browser dispone di una versione aggiornata nella cache

- nessun ritardo nella trasmissione degli oggetti (o utilizzo delle risorse di rete)

■ **client:** specificare la data di creazione dell'oggetto browser

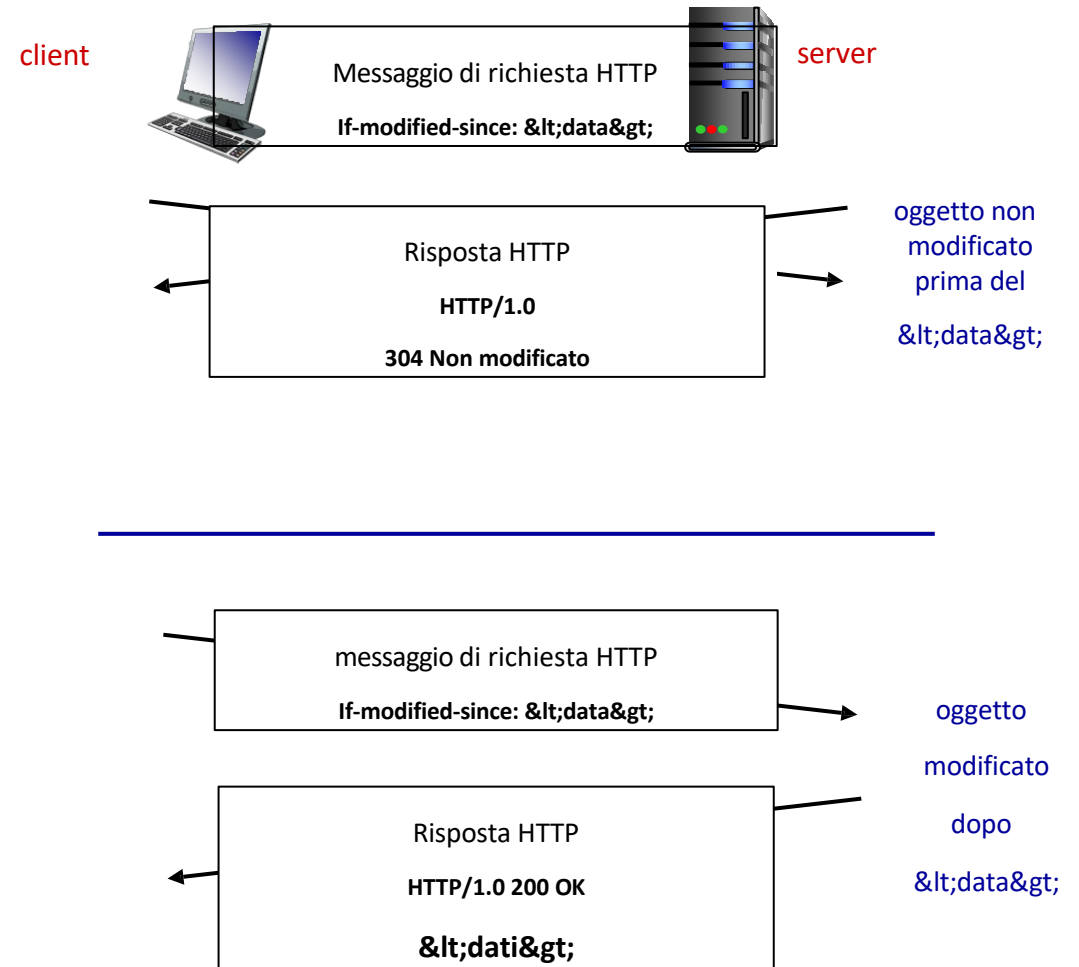
copia memorizzata nella cache nella richiesta HTTP

If-modified-since: <data>;

■ **server:** la risposta non contiene

oggetto se la copia memorizzata nella cache del browser è aggiornata:

HTTP/1.0 304 Non modificato



HTTP/2

Obiettivo principale: riduzione dei ritardi nelle richieste HTTP multi-oggetto

HTTP1.1: introduzione di GET multipli in pipeline su una singola connessione TCP

- il server risponde *in ordine* (FCFS: scheduling first-come-first-served) alle richieste GET
- con FCFS, gli oggetti di piccole dimensioni potrebbero dover attendere la trasmissione (**blocco head-of-line (HOL)**) dietro oggetti di grandi dimensioni
- il recupero delle perdite (ritrasmissione dei segmenti TCP persi) blocca la trasmissione degli oggetti

HTTP/2

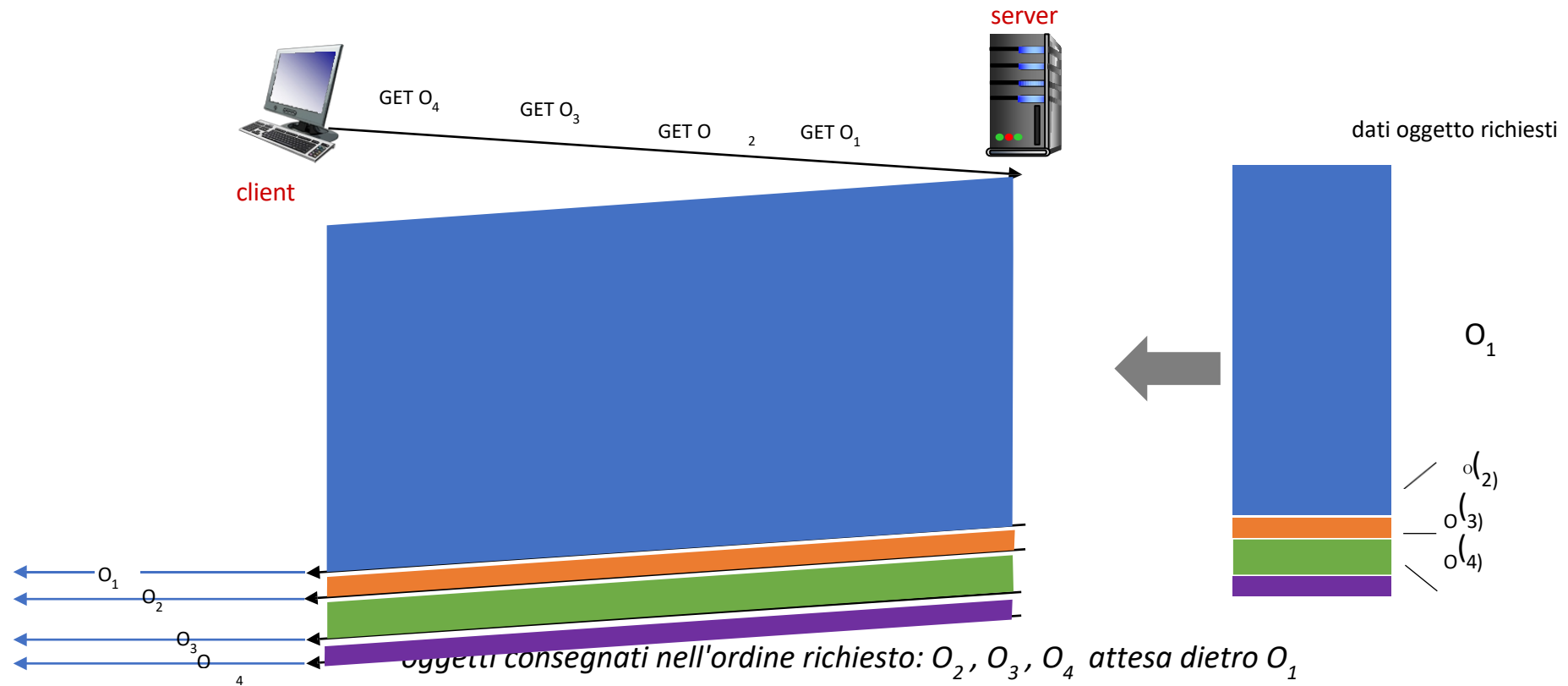
Obiettivo principale: riduzione del ritardo nelle richieste HTTP multi-oggetto

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

- metodi, codici di stato, la maggior parte dei campi dell'intestazione rimangono invariati rispetto a HTTP 1.1
- ordine di trasmissione degli oggetti richiesti basato sulla priorità degli oggetti specificata dal client (non necessariamente FCFS)
- *invio* di oggetti non richiesti al client
- divisione degli oggetti in frame, pianificazione dei frame per mitigare il blocco HOL

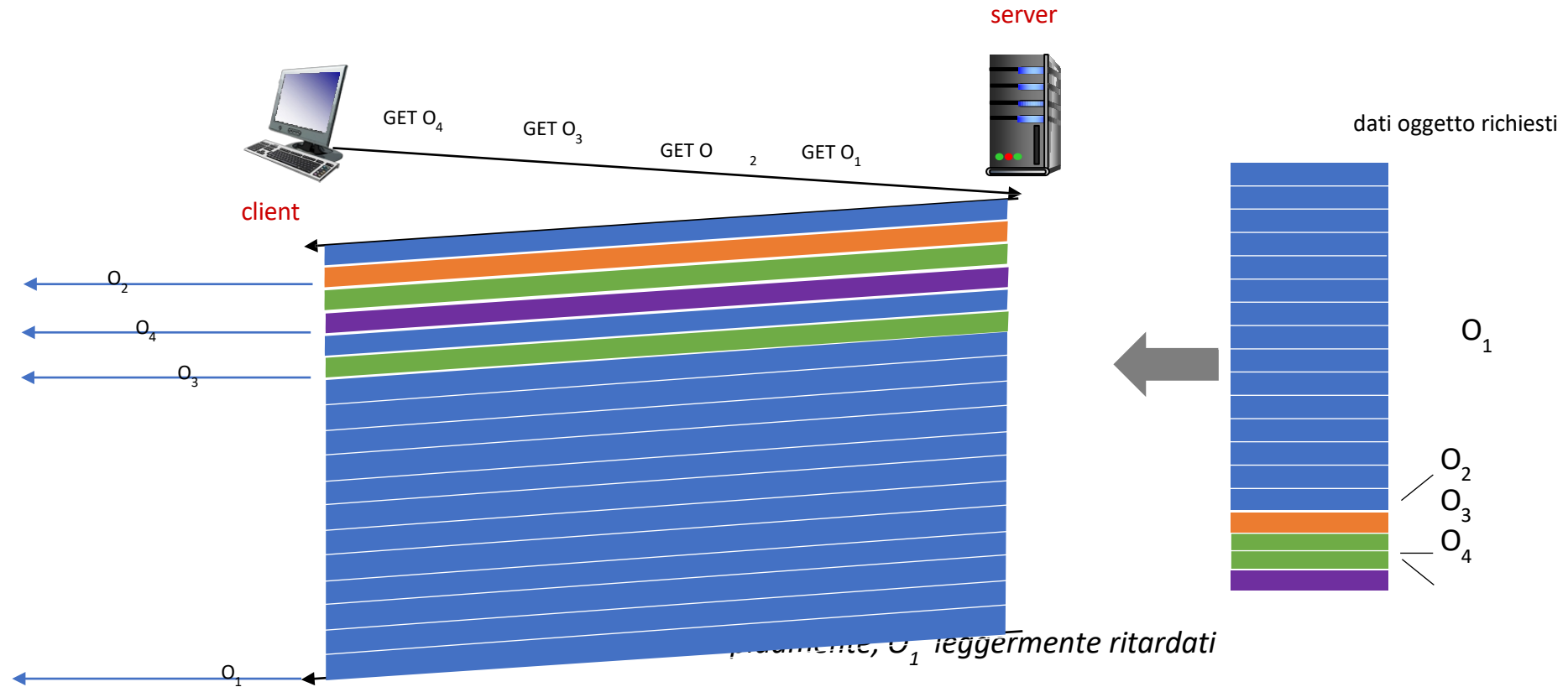
HTTP/2: mitigazione del blocco HOL

HTTP 1.1: il client richiede 1 oggetto di grandi dimensioni (ad esempio, un file video) e 3 oggetti più piccoli



HTTP/2: mitigazione del blocco HOL

HTTP/2: oggetti suddivisi in frame, trasmissione dei frame intercalata



HTTP/2 a HTTP/3

HTTP/2 su singola connessione TCP significa:

- il recupero dalla perdita di pacchetti blocca ancora tutte le trasmissioni di oggetti
 - come in HTTP 1.1, i browser hanno un incentivo ad aprire più connessioni TCP parallele per ridurre il blocco e aumentare la velocità complessiva
- nessuna sicurezza sulla connessione TCP standard
- **HTTP/3:** aggiunge sicurezza, controllo degli errori e della congestione per oggetto (più pipelining) su UDP
 - maggiori informazioni su HTTP/3 nel livello di trasporto

Livello applicativo: panoramica

- Principi delle applicazioni di rete
- Web e HTTP
- E-mail, SMTP, IMAP
- Il sistema dei nomi di dominio DNS
- Applicazioni P2P
- Streaming video e reti di distribuzione dei contenuti
- Programmazione socket con UDP e TCP



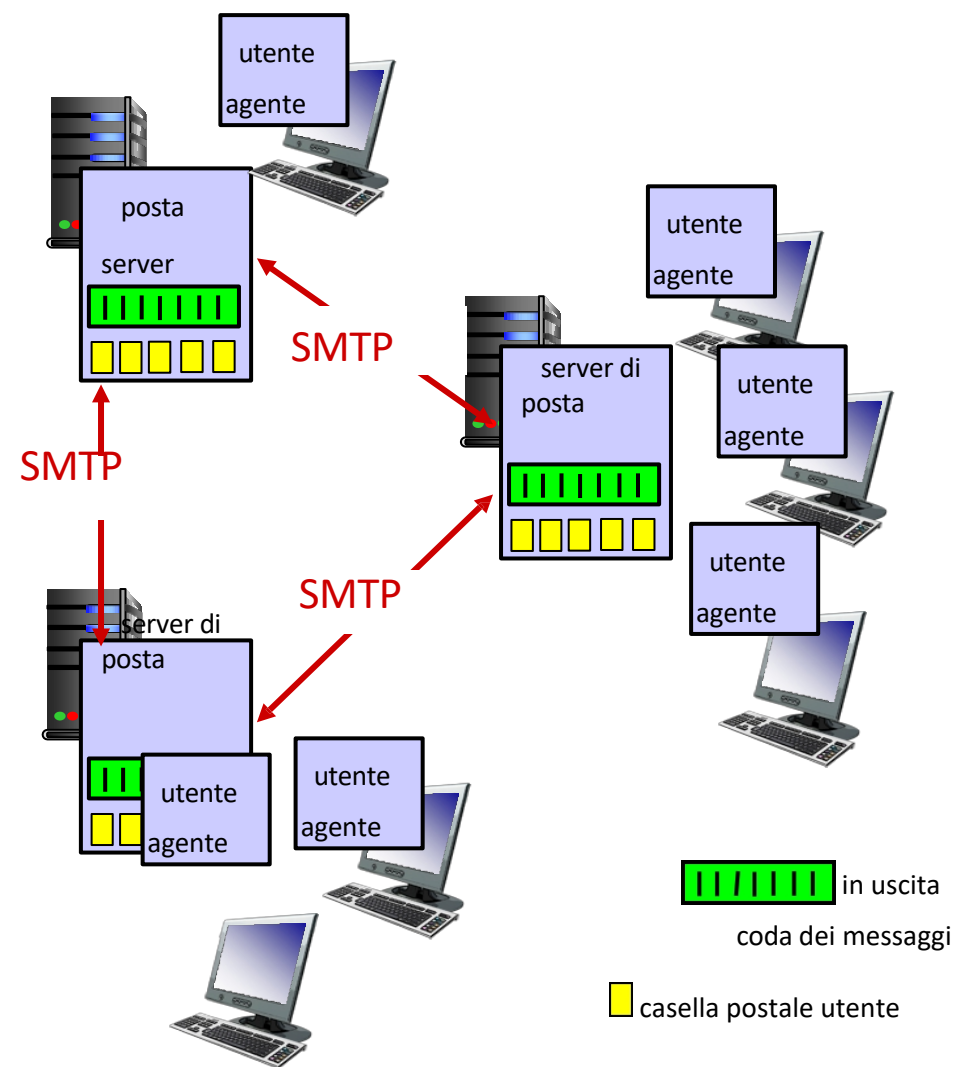
E-mail

Tre componenti principali:

- agenti utente
- server di posta
- protocollo semplice di trasferimento della posta: SMTP

Agente utente

- noto anche come "lettore di posta"
- composizione, modifica, lettura dei messaggi di posta
- ad es. Outlook, client di posta elettronica iPhone
- messaggi in uscita e in entrata memorizzati su server



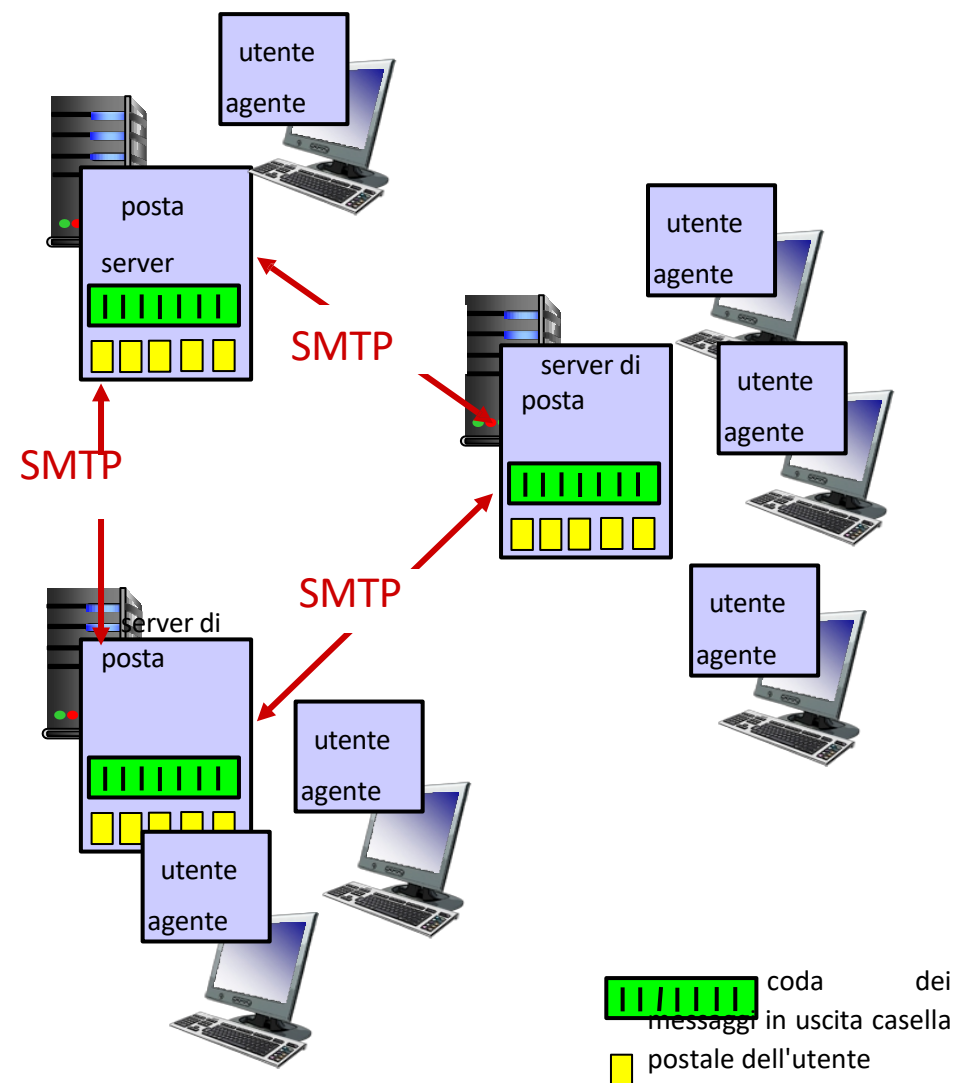
E-mail: server di posta

server di posta:

- *la casella di posta* contiene i messaggi in arrivo per l'utente
- *coda* dei messaggi in uscita (da inviare)

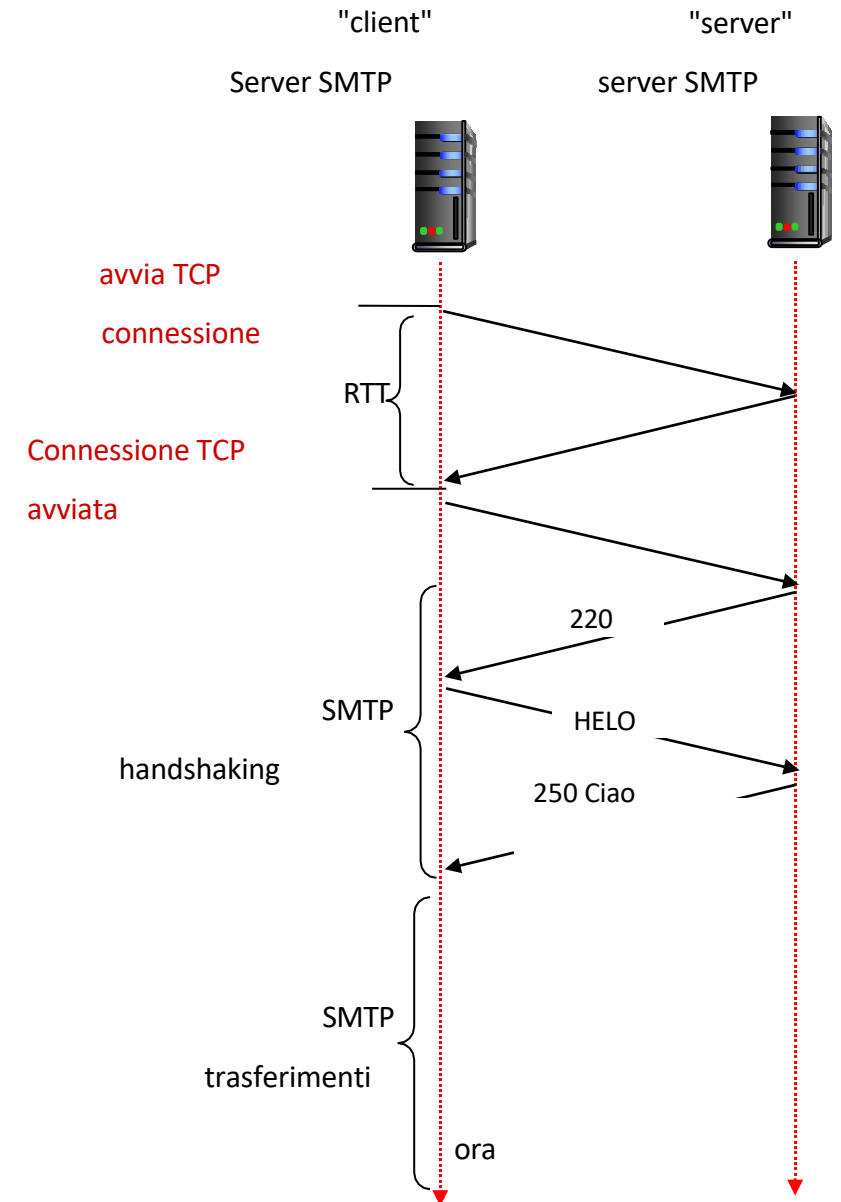
protocollo **SMTP** tra server di posta per inviare messaggi e-mail

- **client**: server di posta in uscita
- **"server"**: server di posta in ricezione



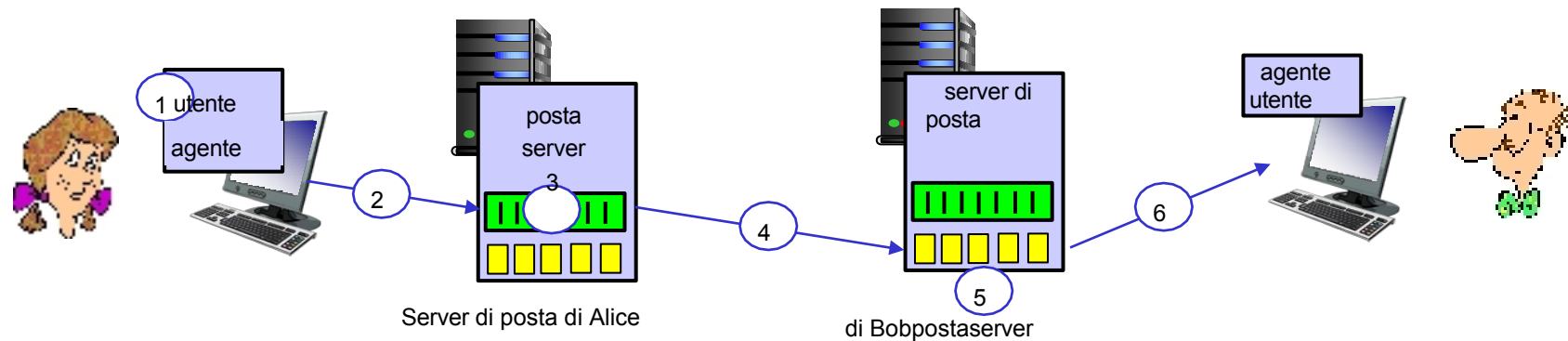
SMTP RFC (5321)

- utilizza il protocollo TCP per trasferire in modo affidabile i messaggi e-mail dal client (server di posta che avvia la connessione) al server, porta 25
 - trasferimento diretto: server di invio (che agisce come client) al server ricevente
- tre fasi di trasferimento
 - Handshaking SMTP (saluto)
 - trasferimento SMTP dei messaggi
 - Chiusura SMTP
- interazione comando/risposta (come HTTP)
 - **comandi**: testo ASCII
 - **risposta**: codice di stato e frase



Scenario: Alice invia un'e-mail a Bob

- 1) Alice utilizza UA per comporre un messaggio e-mail
"a" bob@someschool.edu
- 2) L'UA di Alice invia il messaggio al suo server di posta utilizzando SMTP; il messaggio viene inserito nella coda dei messaggi
- 3) Il lato client SMTP del server di posta apre una connessione TCP con il server di posta di Bob
- 4) Il client SMTP invia il messaggio di Alice tramite la connessione TCP
- 5) Il server di posta di Bob inserisce il messaggio nella casella di posta di Bob
- 6) Bob richiama il suo agente utente per leggere il messaggio



Esempio di interazione SMTP

```
S: 220    hamburger.edu
C: HELO    crepes.fr
S: 250     Ciao crepes.fr, piacere                Piacere di
                                                    conoscerti

C: MAIL    DA: <alice@crepes.fr>;
S: 250     alice@crepes.fr... Mittente            ok

C: RCPT    TO: <bob@hamburger.edu >;
S: 250bob@hamburger.edu ... Destinatario ok C: DATI

S: 354 Inserisci l'indirizzo e-mail, terminando con un punto su una riga separata
C: Ti piace il ketchup? C: E i
sottaceti?

C: .
S: 250 Messaggio accettato per la consegna
C: QUIT
S: 221 hamburger.edu chiusura connessione
```

SMTP: osservazioni

confronto con HTTP:

- HTTP: client pull
- SMTP: client push
- entrambi hanno interazione comando/risposta ASCII, codici di stato
- HTTP: ogni oggetto è incapsulato nel proprio messaggio di risposta
- SMTP: più oggetti inviati in un messaggio multiparte
- SMTP utilizza connessioni persistenti
- SMTP richiede che il messaggio (intestazione e corpo) sia in ASCII a 7 bit
- Il server SMTP utilizza CRLF.CRLF per determinare la fine del messaggio

Formato dei messaggi di posta

SMTP: protocollo per lo scambio di messaggi e-mail, definito nella RFC 5321 (come la RFC 7231 definisce l'HTTP)

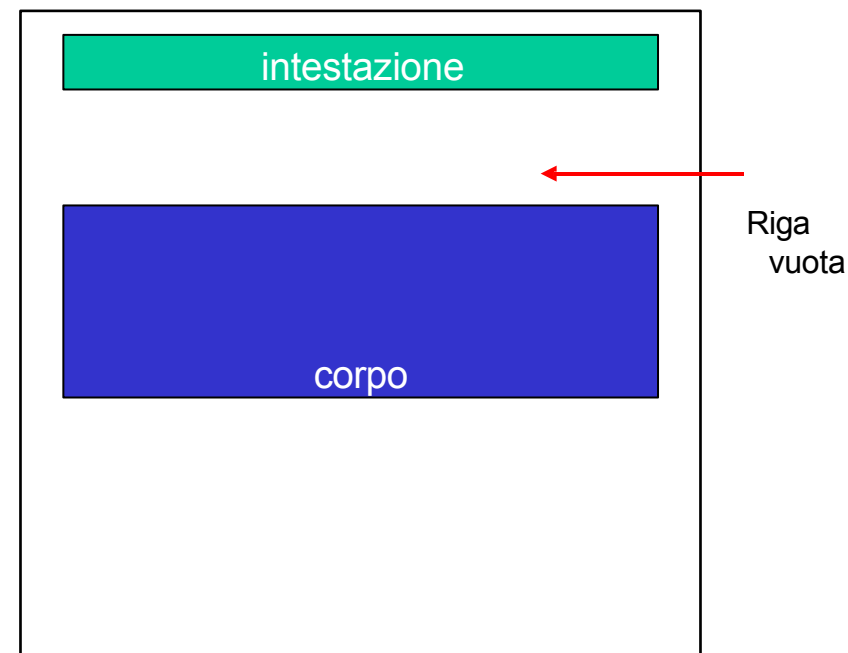
RFC 2822 definisce *la sintassi* per i messaggi e-mail stessi (come HTML definisce la sintassi per i documenti web)

- righe dell'intestazione, ad esempio

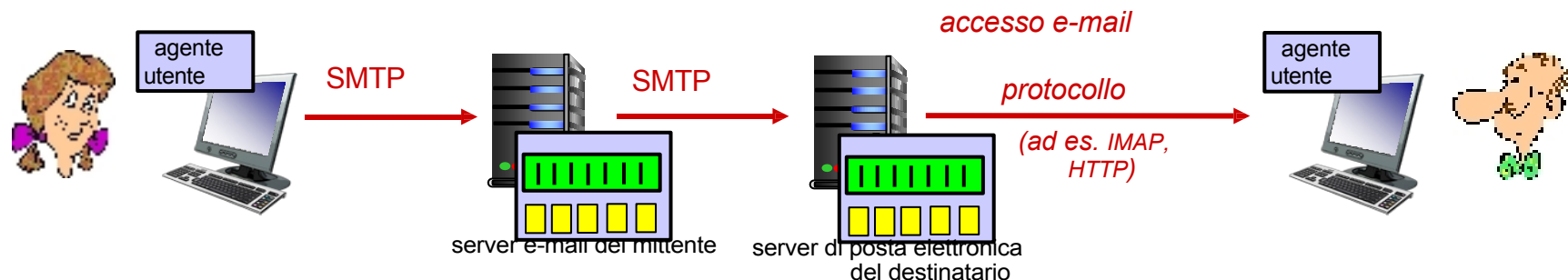
- A:
- Da:
- Oggetto:

queste righe, all'interno dell'area del corpo del messaggio e-mail, diverse dai comandi SMTP MAIL FROM: e RCPT TO:!

- Corpo: il "messaggio", solo caratteri ASCII



Recupero delle e-mail: protocolli di accesso alla posta



- **SMTP:** consegna/archiviazione dei messaggi e-mail sul server del destinatario
- protocollo di accesso alla posta: recupero dal server
 - **IMAP:** Internet Mail Access Protocol [RFC 3501]: messaggi archiviati sul server, IMAP consente il recupero, l'eliminazione e la creazione di cartelle dei messaggi archiviati sul server
- **HTTP:** Gmail, Hotmail, Yahoo!Mail, ecc. forniscono un'interfaccia web basata su SMTP (per l'invio) e IMAP (o POP) per il recupero dei messaggi e-mail

Livello applicativo: panoramica

- Principi delle applicazioni di rete
 - Web e HTTP
 - E-mail, SMTP, IMAP
 - Il sistema dei nomi di dominio DNS
- Applicazioni P2P
 - Streaming video e reti di distribuzione dei contenuti
 - Programmazione socket con UDP e TCP



DNS: Sistema dei nomi di dominio

persone: molte identificatori:

- SSN, nome, numero di passaporto

Host Internet, router:

- indirizzo IP (32 bit) - utilizzato per indirizzare i datagrammi
- "nome", ad esempio cs.umass.edu -

utilizzato dagli esseri umani

D: come mappare l'indirizzo IP e il nome e viceversa?

Sistema dei nomi di dominio (DNS):

- *database distribuito* implementato in una gerarchia di molti *server di nomi*
- *protocollo a livello di applicazione*: host e server DNS comunicano per *risolvere* i nomi (traduzione indirizzo/nome)
 - *nota*: funzione Internet fondamentale, **implementata come protocollo a livello di applicazione**
 - Complessità ai "margin" della rete

DNS: servizi, struttura

Servizi DNS:

- traduzione da nome host a indirizzo IP
- alias host
 - nomi canonici, nomi alias
- alias del server di posta
- distribuzione del carico
 - server Web replicati: molti indirizzi IP corrispondono a un unico nome

D: Perché non centralizzare il DNS?

- singolo punto di errore
- volume di traffico
- database centralizzato distante
- manutenzione

R: Non è scalabile!

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

Riflessioni sul DNS

enorme database distribuito:

- ~ miliardi di record, ciascuno dei quali

gestisce molti *triloni* di query al giorno:

- *molte* più letture che scritture
- *le prestazioni contano*: quasi tutte le transazioni Internet interagiscono con il DNS - ogni millisecondo conta!

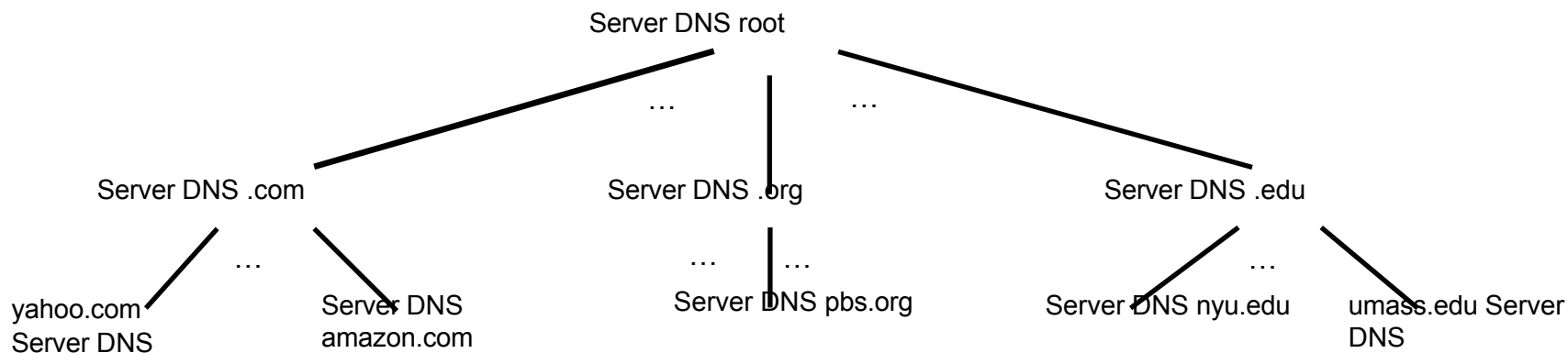
decentralizzato dal punto di vista organizzativo e fisico:

- milioni di organizzazioni diverse responsabili dei propri record

"a prova di proiettile": affidabilità, sicurezza



DNS: un database distribuito e gerarchico



Root

Dominio di primo livello

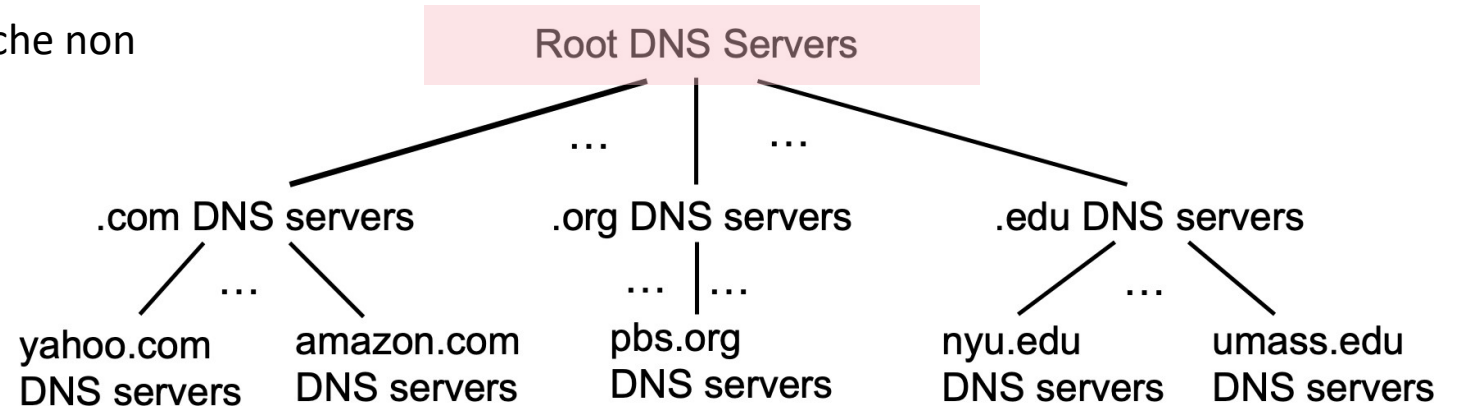
Autoritativo

Il client richiede l'indirizzo IP per www.amazon.com; 1° approssimazione:

- il client interroga il server root per trovare il server DNS .com
- il client interroga il server DNS .com per ottenere il server DNS amazon.com
- Il client interroga il server DNS amazon.com per ottenere l'indirizzo IP di www.amazon.com

DNS: server dei nomi root

- ufficiali, di ultima istanza per nome server che non riescono a risolvere il nome

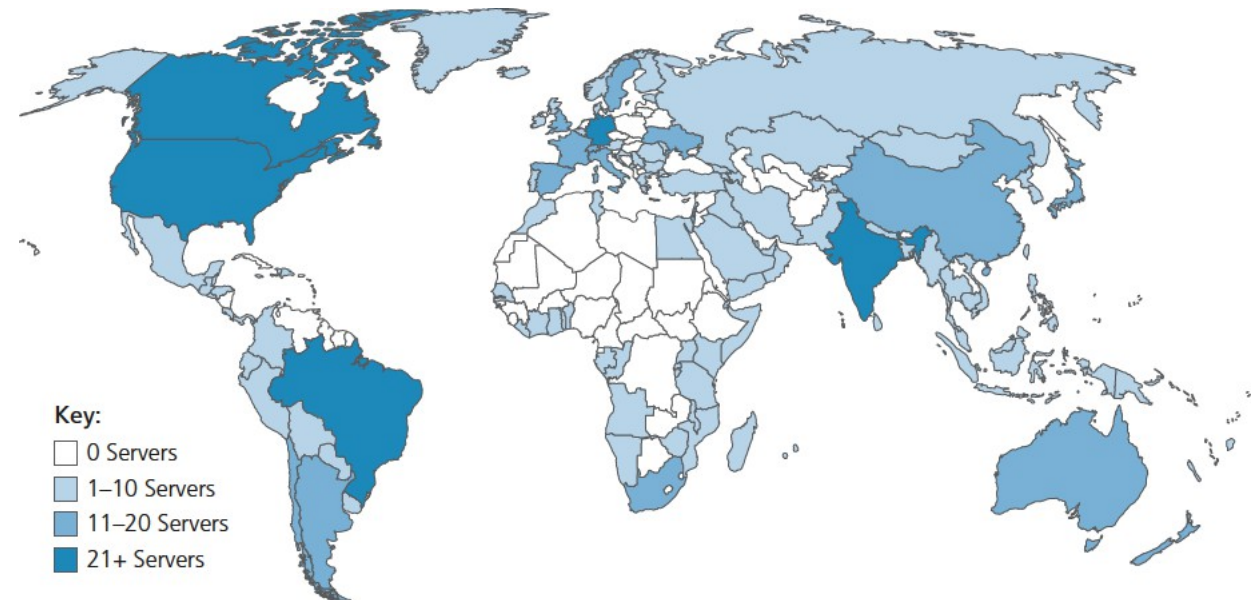


DNS: server dei nomi root

- ufficiali, contatto di ultima istanza dai server dei nomi che non riescono a risolvere il nome
- funzione Internet *incredibilmente importante*
 - Internet non potrebbe funzionare senza di essa!
 - DNSSEC: garantisce la sicurezza (autenticazione, integrità dei messaggi)
- ICANN (Internet Corporation for Assigned Names and Numbers) gestisce il dominio DNS root

13 "server" di nomi root logici in tutto il mondo ogni "server" replicato

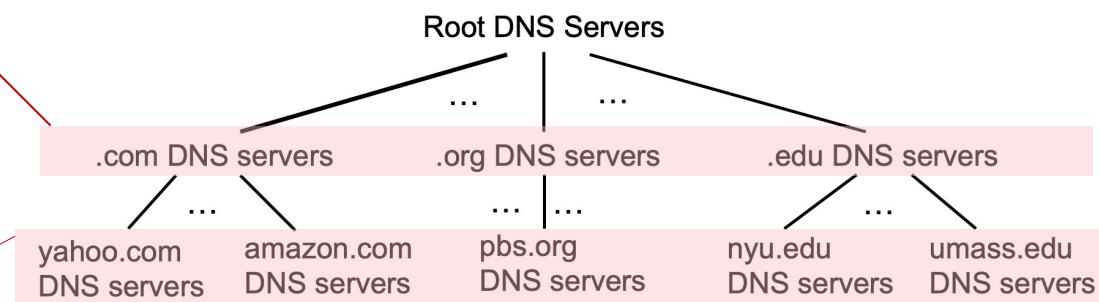
molte volte (~200 server negli Stati Uniti)



Dominio di primo livello e server autoritativi

Server di dominio di primo livello (TLD):

- responsabili di .com, .org, .net, .edu, .aero, .jobs, .museums e tutti i domini di primo livello
domini nazionali, ad esempio: .cn, .uk, .fr, .ca, .jp
- Network Solutions: registro autorevole per i TLD .com, .net
- Educause: TLD .edu



server DNS autorevoli:

- server DNS propri dell'organizzazione, che forniscono mappature autorevoli da nome host a IP
per gli host denominati dell'organizzazione
- possono essere gestiti dall'organizzazione o dal fornitore di servizi

Server DNS locali

- quando l'host effettua una query DNS, questa viene inviata al proprio server DNS *locale*
 - Il server DNS locale restituisce una risposta, rispondendo:
 - dalla sua cache locale delle recenti coppie di traduzioni nome-indirizzo (possibilmente non aggiornate!)
 - inoltrando la richiesta nella gerarchia DNS per la risoluzione
 - ogni ISP ha un server DNS locale; per trovare il tuo:
 - MacOS: `% scutil --dns`
 - Windows: `>ipconfig /all`
- il server DNS locale non appartiene strettamente alla gerarchia

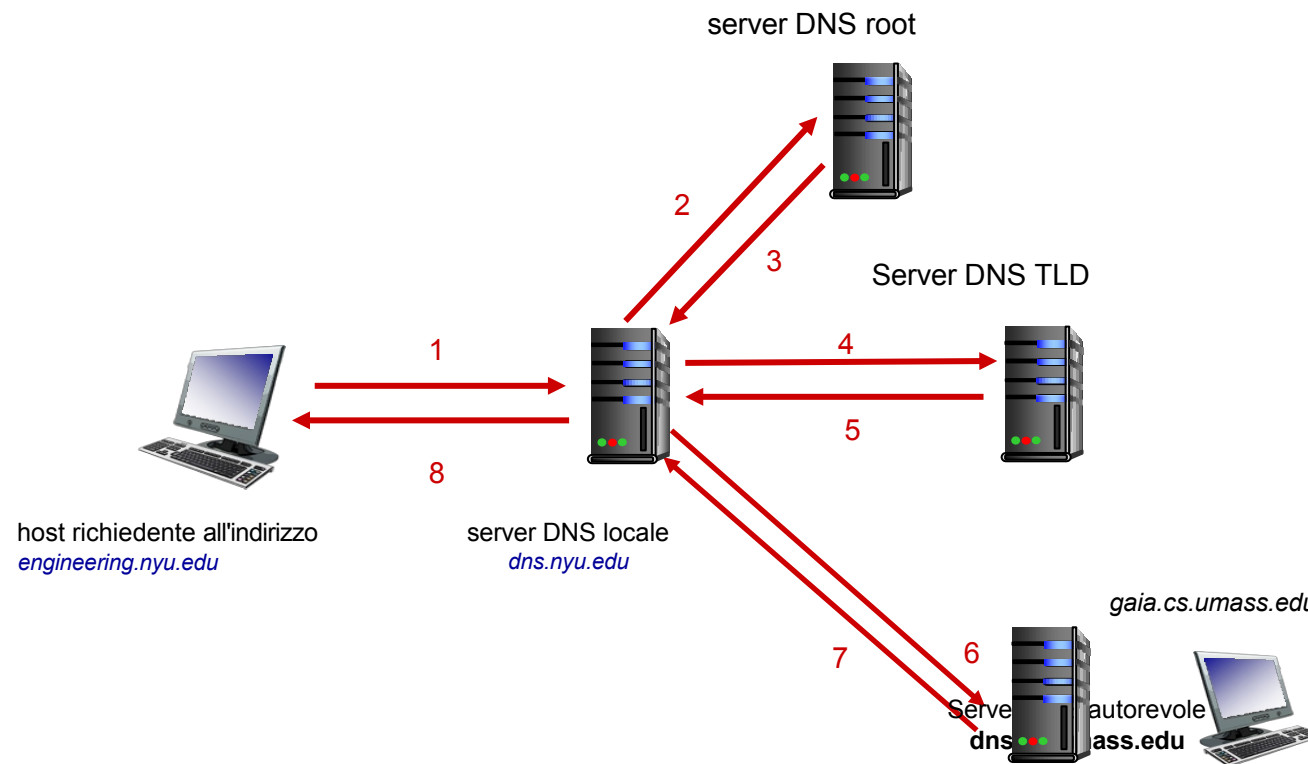
Risoluzione dei nomi DNS: query iterata

Esempio: host su `engineering.nyu.edu`

richiede l'indirizzo IP per `gaia.cs.umass.edu`

Query iterata:

- il server contattato risponde con il nome del server da contattare
- "Non conosco questo nome, ma chiedi a questo server"



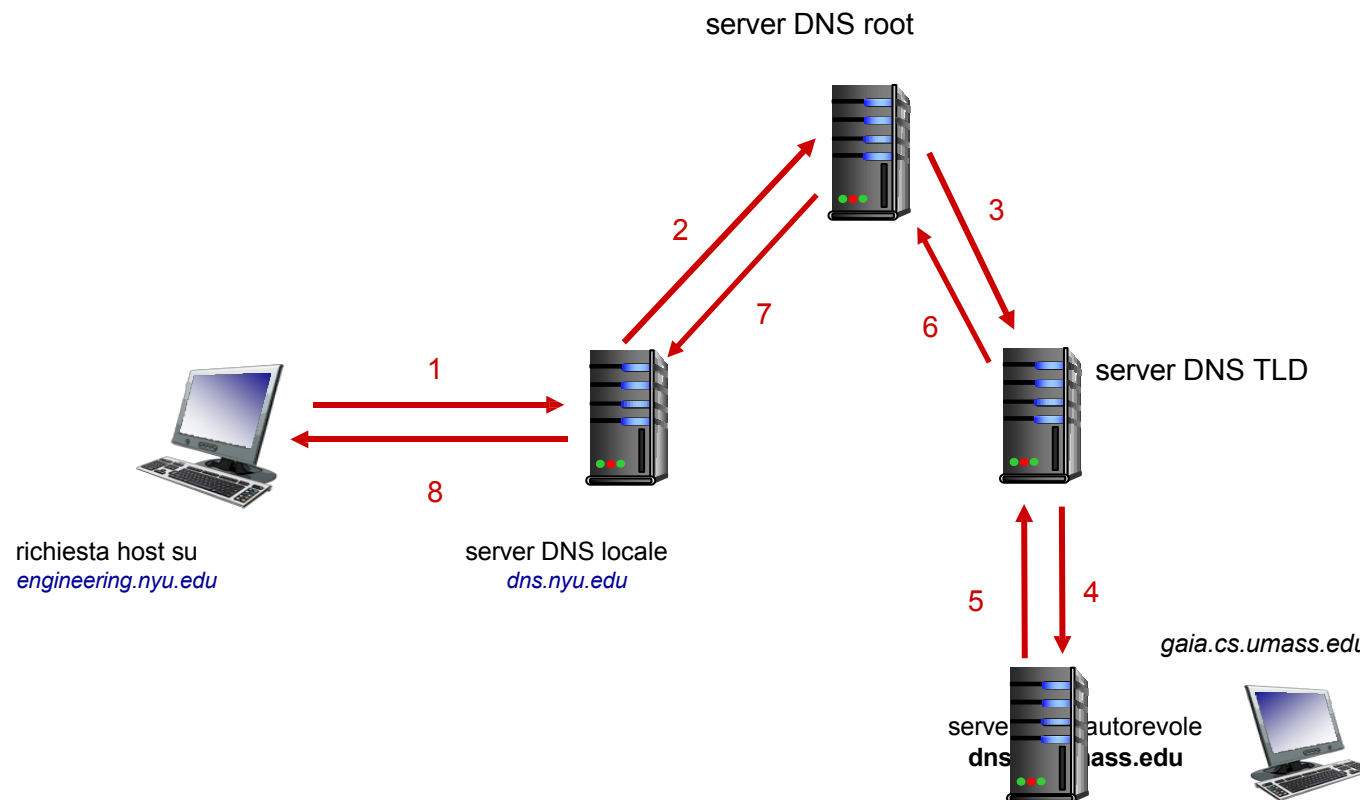
Risoluzione del nome DNS: query ricorsiva

Esempio: host su engineering.nyu.edu

richiede l'indirizzo IP per gaia.cs.umass.edu

Query ricorsiva:

- pone l'onere del nome risoluzione sul server dei nomi contattato
- carico pesante ai livelli superiori della gerarchia?



Memorizzazione delle informazioni DNS

- una volta che un server dei nomi apprende la mappatura, la *memorizza* nella cache e restituisce *immediatamente* la mappatura memorizzata in risposta a una query
 - la memorizzazione nella cache migliora i tempi di risposta
 - Le voci della cache scadono (scompaiono) dopo un certo periodo di tempo (TTL)
 - I server TLD vengono in genere memorizzati nella cache nei server dei nomi locali
- le voci memorizzate nella cache potrebbero *non* essere *aggiornate*
 - se l'host denominato cambia indirizzo IP, potrebbe non essere noto a tutta la rete Internet fino alla scadenza di tutti i TTL!
 - *traduzione nome-indirizzo con il massimo impegno!*

Record DNS

DNS: database distribuito che memorizza i record delle risorse (RR)

Formato RR: (nome, valore, tipo, ttl)

tipo=A

- il nome è il nome host
- il valore è l'indirizzo IP

tipo=NS

- il nome è il dominio (ad esempio, foo.com)
- il valore è il nome host del server dei nomi autoritativo per questo dominio

tipo=CNAME

- il nome è il nome alias per un nome "canonico" (il vero) nome
- www.ibm.com è in realtà servereast.backup2.ibm.com
- il valore è il nome canonico

tipo=MX

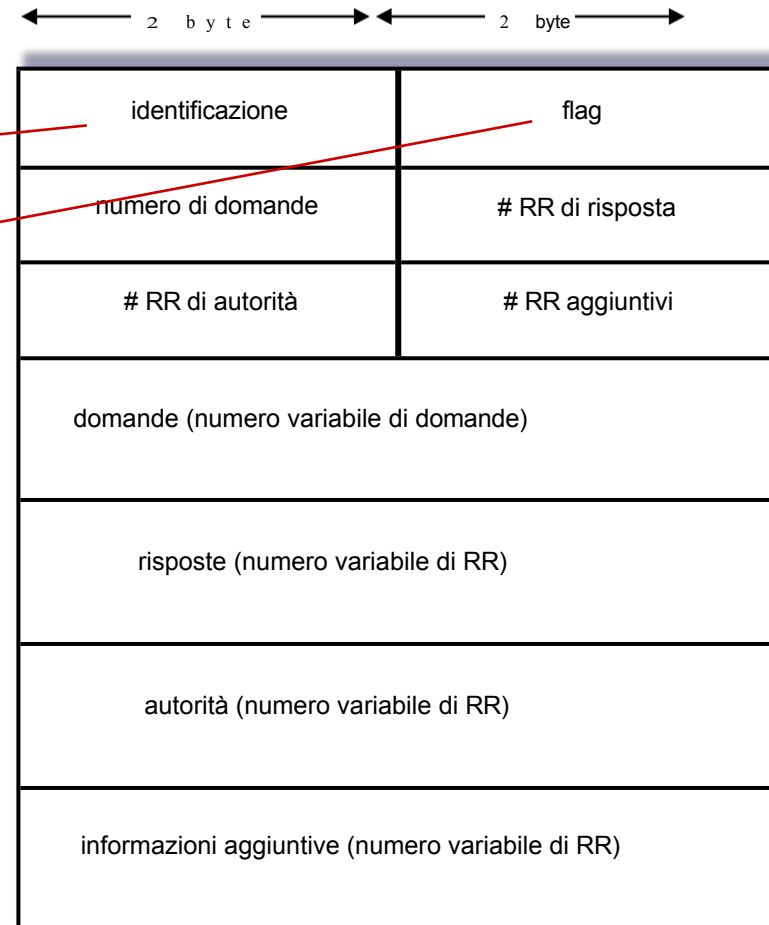
- il valore è il nome del server di posta SMTP
server associato al nome

messaggi del protocollo DNS

I messaggi *di query* e *risposta* DNS hanno entrambi *lo* stesso *formato*:

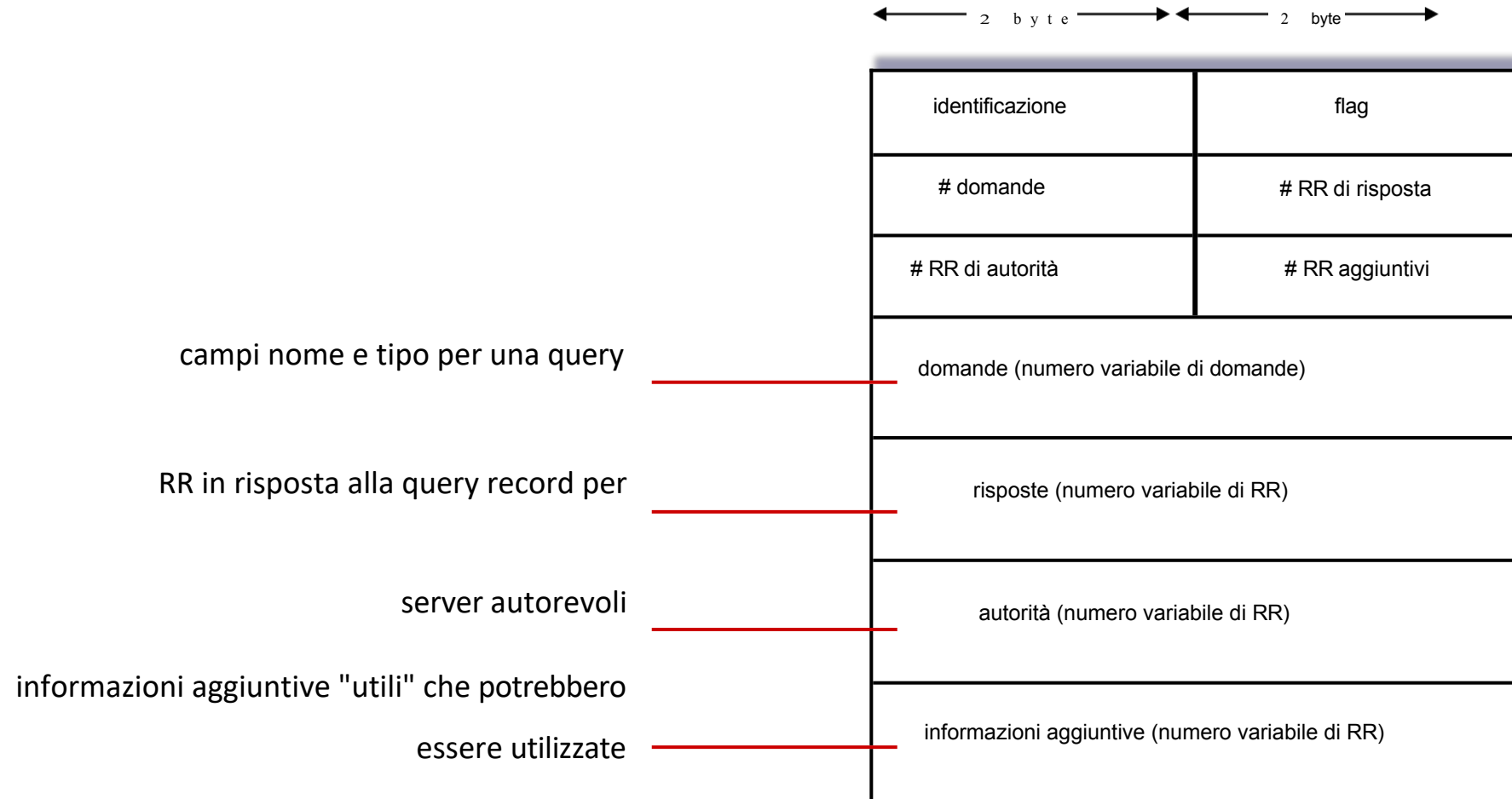
intestazione del messaggio:

- **identificazione:** numero a 16 bit per la query, la risposta alla query utilizza lo stesso numero
- **flag:**
 - query o risposta
 - ricorsione desiderata
 - ricorsione disponibile
 - risposta autorevole



Messaggi del protocollo DNS

I messaggi *di query* e *risposta* DNS hanno entrambi *lo* stesso *formato*:



Inserire le proprie informazioni nel DNS

esempio: nuova startup "Network Utopia"

- registrare il nome networkutopia.com presso *un registrar DNS* (ad es. Network Solutions)
 - fornire nomi, indirizzi IP del server dei nomi autoritativo (primario e secondario)
 - il registrar inserisce NS, A RR nel server TLD .com: (`networkutopia.com`, `dns1.networkutopia.com`, NS) (`dns1.networkutopia.com`, `212.212.212.1`, A)
- creare un server autoritativo localmente con indirizzo IP `212.212.212.1`
 - record di tipo A per `www.networkutopia.com`
 - record MX di tipo A per `networkutopia.com`

Sicurezza DNS

Attacchi DDoS

- bombardare i server root con traffico
 - finora senza successo
 - filtraggio del traffico
 - i server DNS locali memorizzano nella cache gli IP dei server TLD, consentendo di bypassare i server root
- bombardano i server TLD
 - potenzialmente più pericoloso

Attacchi di spoofing

- intercettano le query DNS, restituendo risposte fasulle
 - Avvelenamento della cache DNS
 - RFC 4033: DNSSEC
servizi di autenticazione

Livello applicativo: panoramica

- Principi delle applicazioni di rete
- Web e HTTP
- E-mail, SMTP, IMAP
- Il sistema dei nomi di dominio DNS
- Applicazioni P2P
- Streaming video e reti di distribuzione dei contenuti
- Programmazione socket con UDP e TCP



Streaming video e CDN: contesto

- traffico video in streaming: principale consumatore di larghezza di banda Internet
 - Netflix, YouTube, Amazon Prime: 80% del traffico ISP residenziale (2020)

- *sfida*: scalabilità - come raggiungere

~1 miliardo di utenti?

- *Sfida*: eterogeneità
 - utenti diversi hanno capacità diverse (ad esempio, connessione cablata contro connessione mobile; larghezza di banda elevata contro larghezza di banda ridotta)
- *soluzione*: infrastruttura distribuita a livello di applicazione



Multimedia: video

- video: sequenza di immagini visualizzate a velocità costante
 - ad es. 24 immagini/sec
- immagine digitale: matrice di pixel
 - ogni pixel è rappresentato da bit
- codifica: utilizzare la ridondanza *all'interno* e *tra* le immagini per ridurre il numero di bit utilizzati per codificare l'immagine
 - spaziale (all'interno dell'immagine)
 - temporale (da un'immagine all'altra successiva)

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)



fotogramma i



frame $i+1$

esempio di codifica temporale: invece di inviare il frame completo a $i+1$,

inviare solo le differenze dal frame i

Multimedia: video

- **CBR: (velocità di trasmissione costante):** velocità di codifica video fissa
- **VBR: (bit rate variabile):** la velocità di codifica video varia al variare della quantità di codifica spaziale e temporale
- **esempi:**
 - MPEG 1 (CD-ROM) 1,5 Mbps
 - MPEG2 (DVD) 3-6 Mbps
 - MPEG4 (spesso utilizzato nell'Internet, 64 Kbps – 12 Mbps)

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



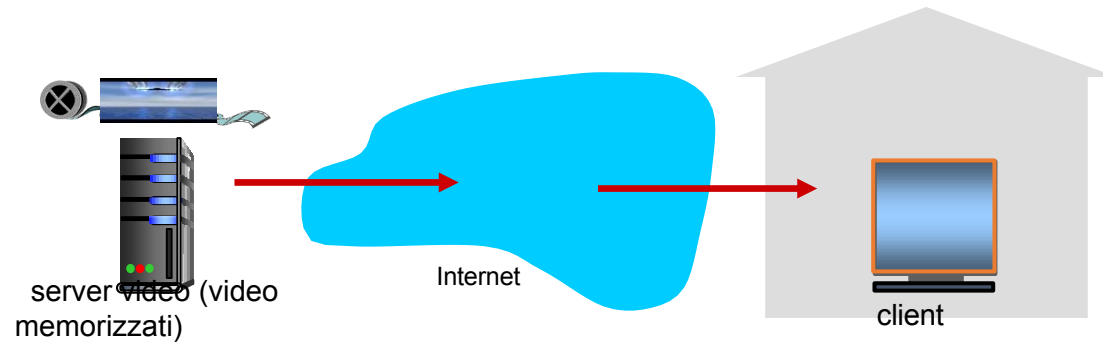
frame i

codifica temporale esempio: invece di inviare il frame completo a $i+1$, inviare solo le differenze rispetto al frame i



Streaming di video memorizzati

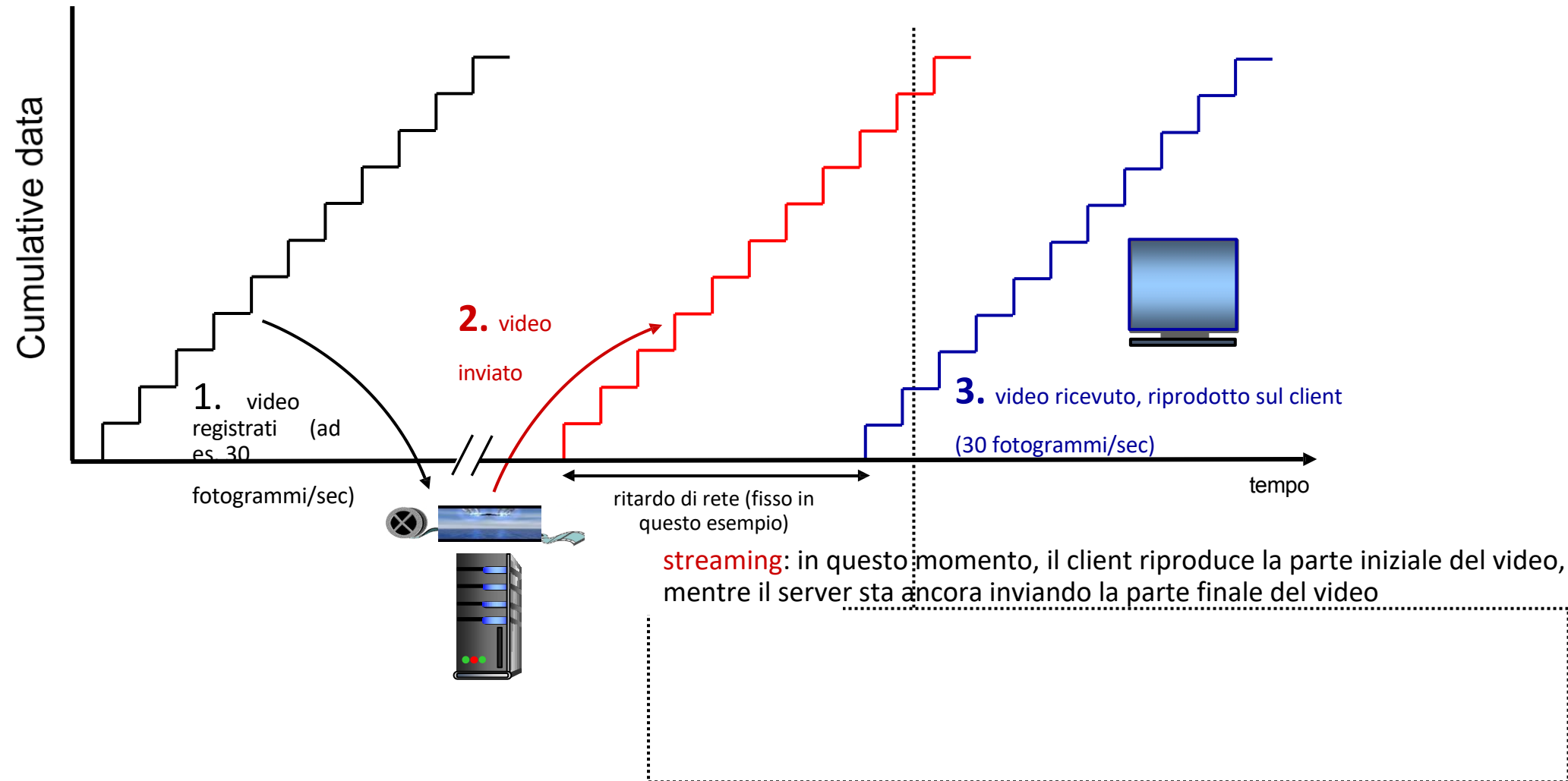
scenario semplice:



Sfide principali:

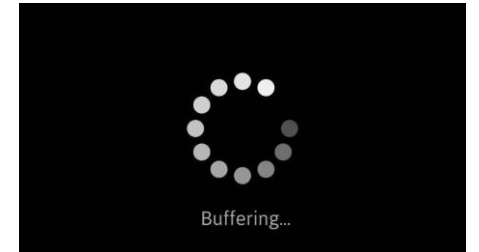
- la larghezza di banda dal server al client *varierà* nel tempo, con livelli di congestione della rete mutevoli (interni, rete di accesso, nucleo della rete, server video)
- la perdita di pacchetti e i ritardi dovuti alla congestione ritarderanno la riproduzione o comprometteranno la qualità video

Streaming di video memorizzati

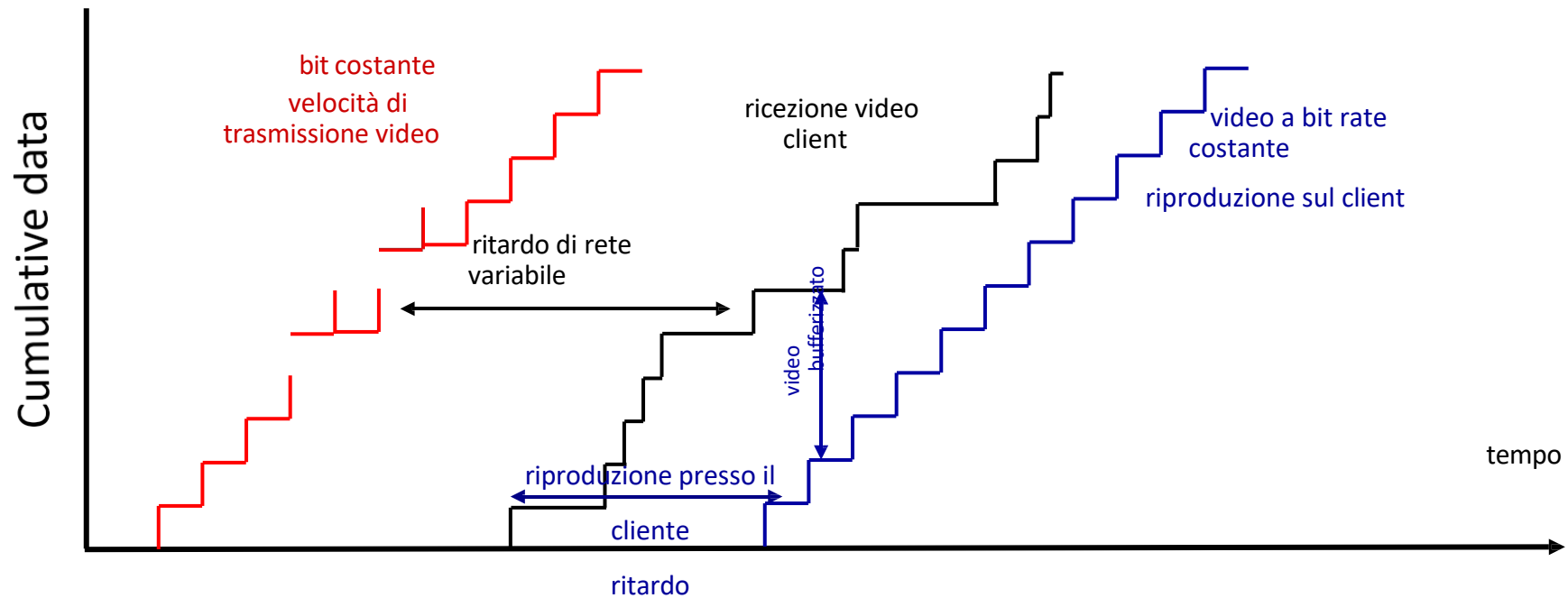


Streaming di video memorizzati: sfide

- **vincolo di riproduzione continua**: durante la riproduzione del video sul client, la tempistica di riproduzione deve corrispondere a quella originale
 - ... ma **i ritardi di rete sono variabili** (jitter), quindi sarà necessario **un buffer lato client** per soddisfare il vincolo di riproduzione continua
- Altre sfide:
 - interattività del client: pausa, avanzamento rapido, riavvolgimento, salto all'interno del video
 - i pacchetti video potrebbero andare persi o essere ritrasmessi



Streaming di video memorizzati: buffering della riproduzione



- *buffering lato client e ritardo di riproduzione*: compensazione del ritardo aggiunto dalla rete, jitter di ritardo

Streaming multimediale: DASH

Dinamico, adattivo

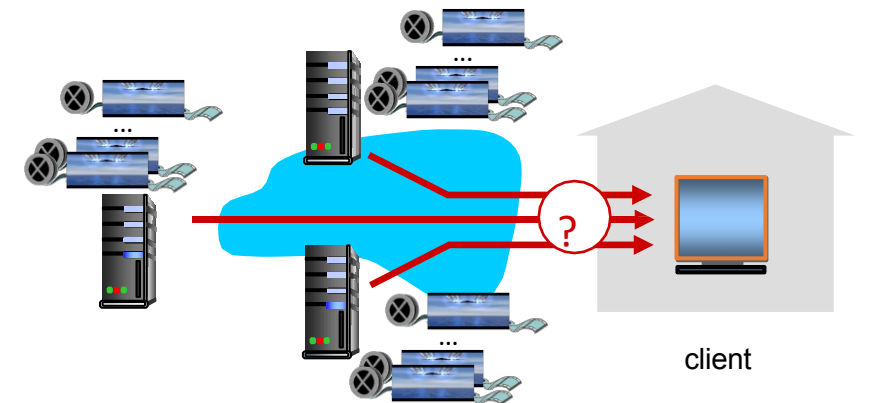
Streaming su HTTP

server:

- divide il file video in più blocchi
- ogni blocco è codificato a velocità diverse
- codifiche a velocità diverse memorizzate in file diversi
- i file vengono replicati in vari nodi CDN
- *file manifest*: fornisce gli URL per i diversi blocchi

client:

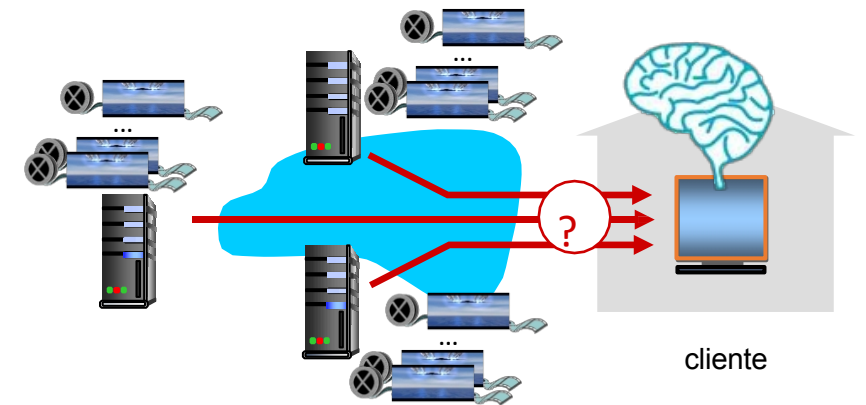
- stima periodicamente la larghezza di banda dal server al client
- consultando il manifesto, richiede un blocco alla volta
 - sceglie la velocità di codifica massima sostenibile data la larghezza di banda attuale
 - può scegliere velocità di codifica diverse in momenti diversi (a seconda della larghezza di banda disponibile in quel momento) e da server diversi



Streaming multimediale: DASH

- *"Intelligenza"* presso il cliente: il cliente determina
 - *quando* richiedere un chunk (in modo che non si verifichino esaurimento del buffer o overflow)
 - *quale velocità di codifica* richiedere (maggiore qualità quando è disponibile una maggiore larghezza di banda)
 - *dove* richiedere il chunk (è possibile richiederlo dal server URL "vicino" al client o con elevata larghezza di banda disponibile)

Streaming video = codifica + DASH + buffering di riproduzione



Reti di distribuzione dei contenuti (CDN)

Sfida: come trasmettere contenuti (selezionati tra milioni di video) a centinaia di migliaia di utenti *contemporaneamente*?

- *opzione 1:* un unico "mega-server" di grandi dimensioni
 - unico punto di errore
 - punto di congestione della rete
 - percorso lungo (e potenzialmente congestionato) verso client distanti

... in poche parole: questa soluzione *non è scalabile*

Reti di distribuzione dei contenuti (CDN)

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

■ *opzione 2:* archiviare/fornire più copie dei video in più siti distribuiti geograficamente (*CDN*)

- *Entrare in profondità:* spingere i server CDN in profondità in molte reti di accesso

- vicino agli utenti
- Akamai: 240.000 server distribuiti in oltre 120 paesi (2015)

- *portare a casa:* un numero ridotto (decine) di

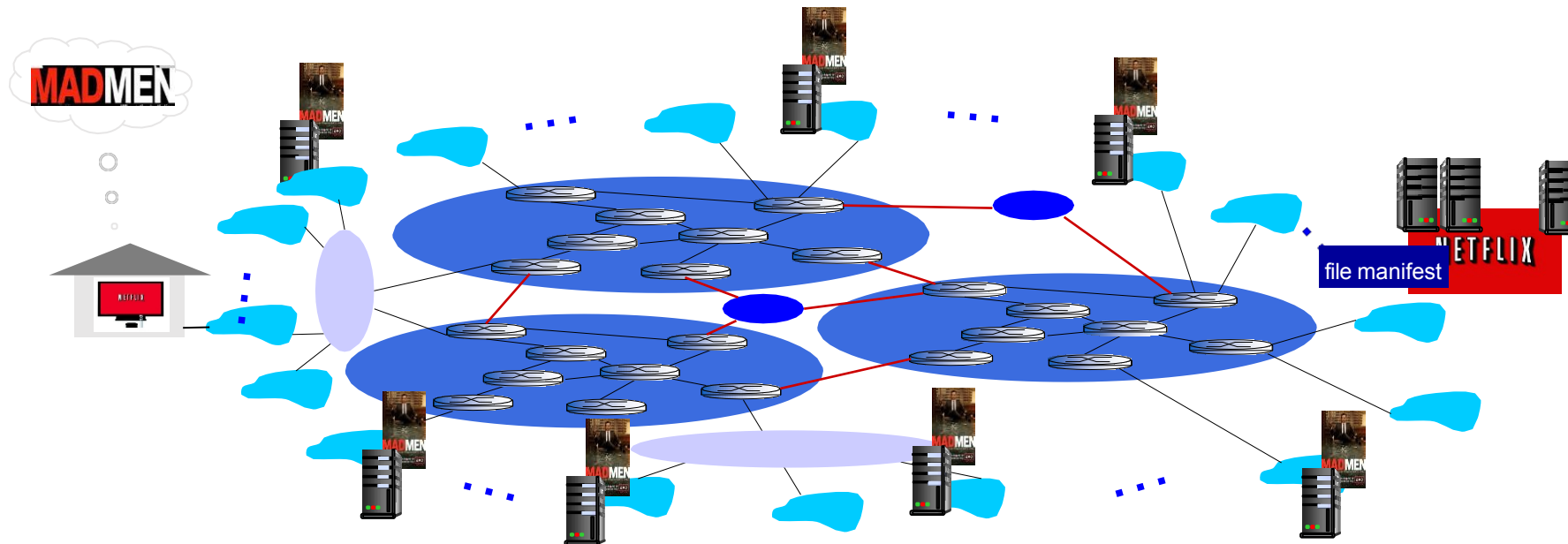
cluster più grandi nei POP vicino alle reti di accesso

- utilizzati da Limelight



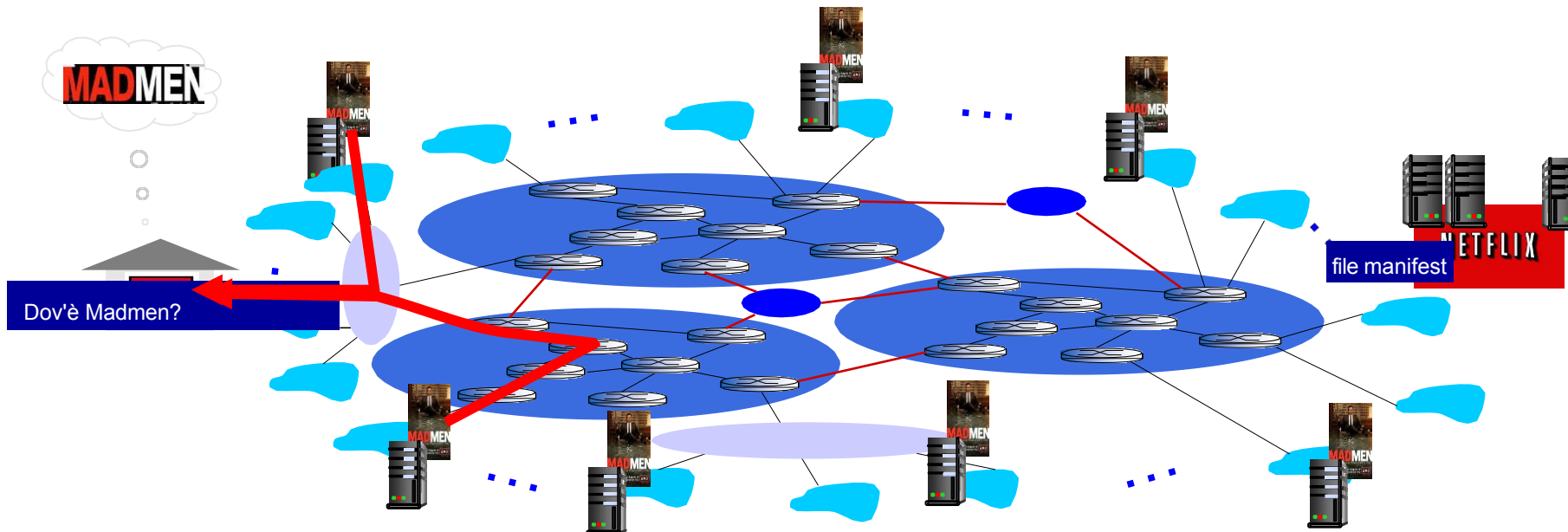
Come funziona Netflix?

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



Come funziona Netflix?

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



Livello applicativo: panoramica

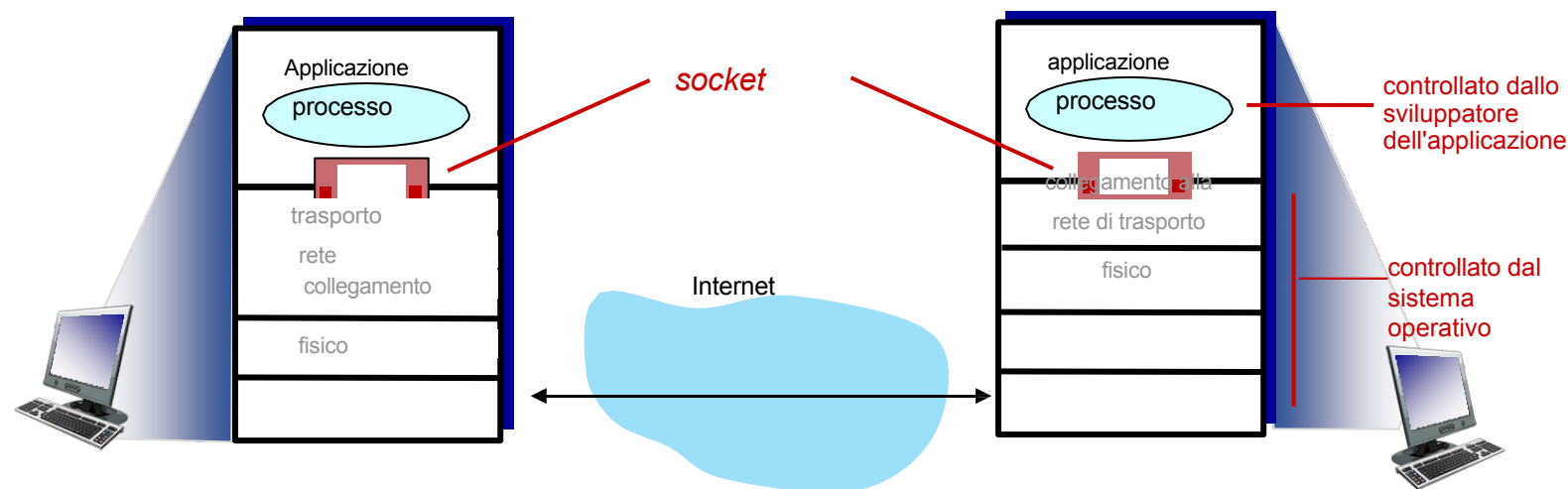
- Principi delle applicazioni di rete
- Web e HTTP
- E-mail, SMTP, IMAP
- Il sistema dei nomi di dominio DNS
- Applicazioni P2P
- Streaming video e reti di distribuzione dei contenuti
- Programmazione socket con UDP e TCP



Programmazione socket

Obiettivo: imparare a creare applicazioni client/server che comunicano tramite socket

Socket: porta tra il processo dell'applicazione e il protocollo di trasporto end-to-end



Programmazione socket

Due tipi di socket per due servizi di trasporto:

- *UDP*: datagramma inaffidabile
- *TCP*: affidabile, orientato al flusso di byte

Esempio di applicazione:

1. il client legge una riga di caratteri (dati) dalla tastiera e invia i dati 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 visualizza la riga sullo schermo

Programmazione socket con UDP

UDP: nessuna "connessione" tra

client e server:

- nessun handshaking prima dell'invio dei dati
- il mittente allega esplicitamente l'indirizzo IP di destinazione e il numero di porta a ciascun pacchetto
- il destinatario estrae l'indirizzo IP e il numero di porta del mittente dal pacchetto ricevuto

UDP: i dati trasmessi possono andare persi o essere ricevuti in ordine casuale Punto di vista dell'applicazione:

- UDP fornisce un trasferimento *inaffidabile* di gruppi di byte ("datagrammi")
tra processi client e server

Interazione socket client/server: UDP



server (in esecuzione su serverIP)

crea socket, porta= x: `serverSocket = socket(AF_INET,SOCK_DGRAM)`

leggere datagramma da `serverSocket`

scrivere la risposta su `serverSocket` specificando l'indirizzo del client e il numero di porta



client

crea socket:

`clientSocket = socket(AF_INET,SOCK_DGRAM)`

Creare datagramma con indirizzo serverIP e porta=x; inviare datagramma tramite `clientSocket`

leggere il datagramma da `clientSocket`

chiudere `clientSocket`

App di esempio: client UDP

Python UDPClient

include Libreria socket di Python	→	da socket import *	serverName = 'hostname'	
		serverPort = 12000		
	→	clientSocket = socket(AF_INET,		
			SOCK_DGRAM)	
crea socket UDP	→	message = input('Inserisci una frase in minuscolo:') →		
		clientSocket.sendto(message.encode(),		
			(serverName, serverPort))	
ottenere l'input dalla tastiera dell'utente	→	modifiedMessage, serverAddress =		
allegare il nome del server e la porta al messaggio; inviare al socket			clientSocket.recvfrom(2048) →	
		print(modifiedMessage.decode())		
leggere i dati di risposta (byte) dal socket		clientSocket.close()		
stampa la stringa ricevuta e chiude il socket				

App di esempio: server UDP

Python UDPServer

	da socket import *	serverPort = 12000
crea socket UDP	→	serverSocket = socket(AF_INET, SOCK_DGRAM)
collega socket al numero di porta locale 12000	→	serverSocket.bind(("", serverPort)) print('Il server è pronto a ricevere')
	→	while True:
ciclo infinito Leggi	→	message, clientAddress = serverSocket.recvfrom(2048) modifiedMessage = message.decode().upper() serverSocket.sendto(modifiedMessage.encode(), clientAddress)
dal socket UDP nel messaggio, ottenendo indirizzo del client (IP e porta del client)		
invia stringa maiuscola a questo client	→	

Programmazione socket con TCP

Il client deve contattare il server

- il processo server deve essere in esecuzione
- il server deve aver creato un socket (porta) che accolga il contatto del client

Il client contatta il server tramite:

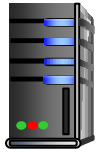
- Creando un socket TCP, specificando l'indirizzo IP e il numero di porta del processo server
- *quando il client crea il socket:* il TCP client stabilisce la connessione con il TCP server

- quando viene contattato dal client, *il TCP del server crea un nuovo socket* per consentire al processo server di comunicare con quel particolare client
 - consente al server di comunicare con più client
 - il numero della porta sorgente del client e l'indirizzo IP vengono utilizzati per distinguere i client (maggiori informazioni nel capitolo 3)

Punto di vista dell'applicazione

Il protocollo TCP garantisce un trasferimento affidabile e ordinato di flussi di byte ("pipe") tra i processi client e server.

Interazione socket client/server: TCP



server (in esecuzione su `hostid`)

crea socket, `porta=x`, per
richiesta in entrata:
`serverSocket = socket()`

attendere la richiesta di
connessione in entrata
`connectionSocket =`

`serverSocket.accept()`

leggere richiesta da
`connectionSocket`

scrivere la risposta su
`connectionSocket`

chiudi
`connectionSocket`

client



crea socket,
connettersi a `hostid`, `porta=x`
`clientSocket = socket()`

invia richiesta utilizzando `clientSocket`

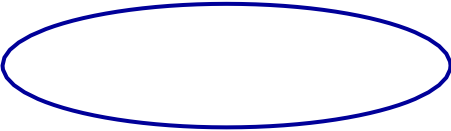
leggi risposta da
`clientSocket` c h i u d i
`clientSocket`

← — — — — — TCP — — — — — →
configurazione connessione

App di esempio: client TCP

Python TCPClient

```
da socket import * serverName = 'servername'
serverPort = 12000
→ clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort)) sentence = input('Inserisci una frase in
minuscolo:') clientSocket.send(sentence.encode())
crea socket TCP per server, porta remota 12000 → fraseModificata = clientSocket.recv(1024)
print ('Dal server:', modifiedSentence.decode())
clientSocket.close()
```



Non è necessario allegare il nome del server e la porta

App di esempio: server TCP

Python TCPServer

crea socket TCP di benvenuto	→	da socket import * portaServer = 12000 serverSocket = socket(AF_INET,SOCK_STREAM) serverSocket.bind(('',serverPort))
il server inizia ad ascoltare le richieste TCP in entrata	→	serverSocket.listen(1) print('Il server è pronto a ricevere')
ciclo infinito	→	while True: connectionSocket, addr = serverSocket.accept()
il server attende su accept() le richieste in arrivo, al ritorno viene creato un nuovo socket	→	sentence = connectionSocket.recv(1024).decode() capitalizedSentence = sentence.upper() connectionSocket.send(capitalizedSentence.encode())
leggi i byte dal socket (ma non l'indirizzo come in UDP)	→	
chiudi connessione a questo client (ma <i>non</i> accogliere il socket)	→	connectionSocket.close()