# Application Layer and Socket Programming

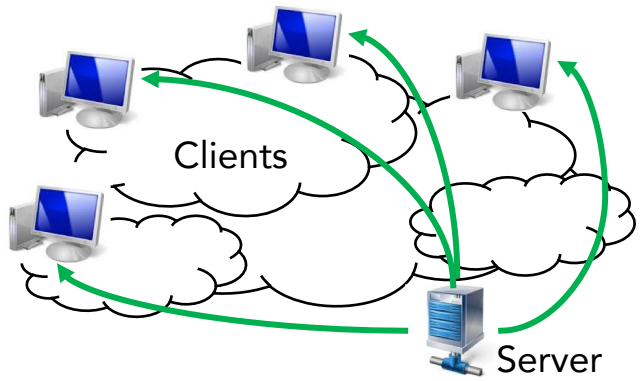
To do …

- App layer principles
- A few protocols
- Socket programming

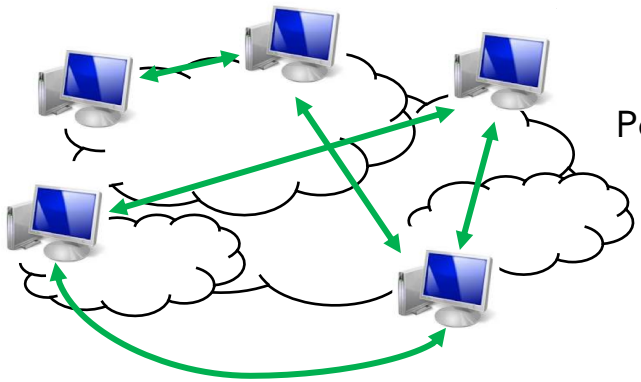
Northwestern

# Network applications

- When developing a new network application
  - Write software that must run on multiple end systems
    - *No need to write code for core routers*
  - that communicate with each other
- How should these applications be organized?



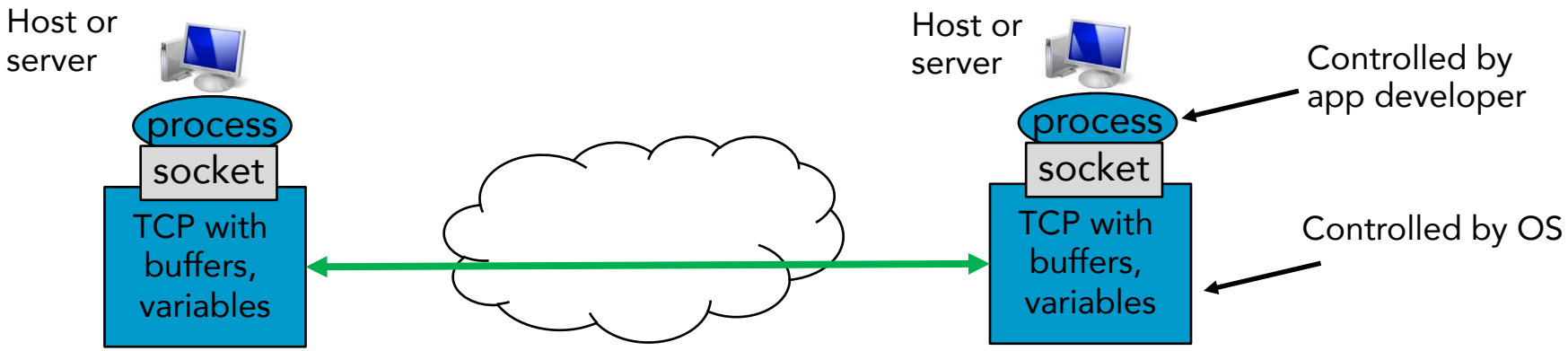

Client-server
- Clients only talk to servers
- Servers are always-on, dedicated machines with well-known addresses




Peer-to-peer
- Minimal or no reliance on servers
- Peers, typically user-controlled, connect directly to provide the service (so, self-scaling)
- Unmanaged … security, performance, reliability?

# Communicating processes

- Processes communicate exchanging messages
  - Each acting, at a given point, as client or server
  - *Client* – the process that initiates the connections
  - *Server* – the one that wats to be connected to
- Processes send/receive mgs through a SW interface: a socket
  - No control over the transport implementation, but can choose which and perhaps fix some parameters



# Communicating processes

- End-point of a connection – a socket
  - Socket address – IP address + port number (16-bit)
  - On the client – ephemeral, assigned by kernel
  - On the server – well-known port (e.g., web 80, SMTP 25)

| Client | Server |
| --- | --- |
| | Socket – create socket<br>Bind – assign address, port<br>Listen – listen for clients |
| Socket – create socket<br>Bind – assign address<br>Connect – connect to listening socket | |
| | Accept - accept connection |
| Both can read/write from the connection<br>Both can call close to end the connection | |

# A simple client

```
/* GetHeadInfo
*/
package main
import (
    "net"
    "os"
    "fmt"
    "io/ioutil"
)

func main() {
   if len(os.Args) != 2 {
       fmt.Fprintf(os.Stderr, "Usage: %s host:port ", os.Args[0])
       os.Exit(1)
   }
   service := os.Args[1]

   tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
   checkError(err)

   conn, err := net.DialTCP("tcp", nil, tcpAddr)
   checkError(err)

   _, err = conn.Write([]byte("HEAD / HTTP/1.0\r\n\r\n"))
   checkError(err)

   result, err := ioutil.ReadAll(conn)
   checkError(err)

   fmt.Println(string(result))
   os.Exit(0)
}
```

```
func checkError(err error) {
   if err != nil {
      fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
      os.Exit(1)
   }
}
```

```
% ./GetHeadInfo www.google.com:80
…
```

# Available Internet transport services

- Many networks provide more than one transport protocol
  - How to choose? That which best matches your application's needs
- What can they offer?
  - Reliable data transfer
    - Not needed for a loss-tolerant app, but good otherwise
  - Throughput
    - Available throughput can fluctuate; minimum throughput guarantees?
  - Timing
    - Timing guarantees (like max delay)
  - Security
    - Encrypt data, can check data integrity or end-point authentication

# Transport Services on the Internet

- Two main transport protocols – TCP and UDP
  - TCP
    - Connection-oriented, full-duplex, reliable transfer service
    - Includes a congestion control mechanism
  - UDP
    - Connectionless, lightweight transport, minimal service
    - No congestion control
- What you don't get
  - No throughput or timing guarantees
  - *What wait? How can we run time-sensitive apps? You can, there are just no timing guarantees*

# Some network applications and their protocols

- Application-layer protocol  defines
  - Type of msgs exchanged
  - Syntax of the various msg types
  - Semantics of the fields
  - Rules for determining when and how a process sends/responds to msgs
- Some protocols are defined in RFCs (e.g., HTTP, RFC 2616),
- Others are proprietary (e.g., Skype)
- Next we'll look at a few examples
  - Web and HTTP, eMail and SMPT, DNS, P2P

# The Web and HTTP

- In the early 90s, a new app – WWW – caught the public's eye
  - Part of the appeal, an 'on-demand' service
    - Unlike broadcast TV or radio – what you want, when you want it
  - Easy to access, to publish, to navigate and to get tangled up
- HyperText Transfer Protocol (HTTP)
  - Web's application-layer protocol (RFC 1945, RFC 26615), runs over TCP
  - A web page made of objects – a base page and several referenced objects with the objects' URLs (uniform resource locator)
  - URLs have the object's hosting server and the object's pathname

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

# HTTP

- Client makes a request, and server sends a response
- Request specifies
  - A human-readable header with: URL, method, some optional headers
  - An optional body, storing raw data (bytes)
- Response includes
  - A human-readable header with: response code, some optional headers
  - An optional body
- HTTP is stateless – server remembers nothing about past requests from this client; request must be self-contained
  - Stateless protocols are simpler and easier to scale, any of multiple servers can reply

# Persistent and non-persistent connections

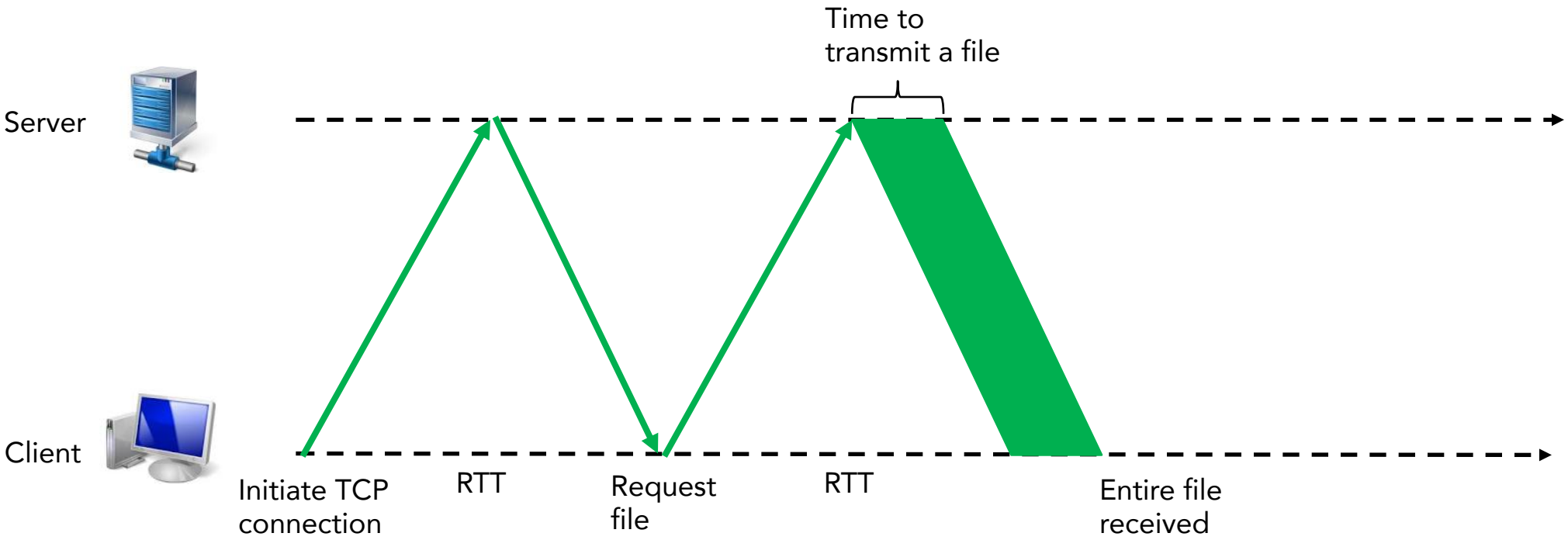
- Client and servers may need to communicate for a while
  - When using TCP, do you want a connection per request/response or do you want them all over the same connection?
- HTTP with non-persistent connections
  - Client initiate a TCP connection
  - Client send an HTTP request
  - Server process request, encapsulate and sends response
  - Server tells TCP to close TCP connection (once done)
  - Client gets response, TCP connection is closed …

Web page with a base document + 10 objects?
*Repeat 11 times*

- This could be done serially – 10 TCP connections one after the other – or with some of them in parallel

# Time to request and receive an HTML file

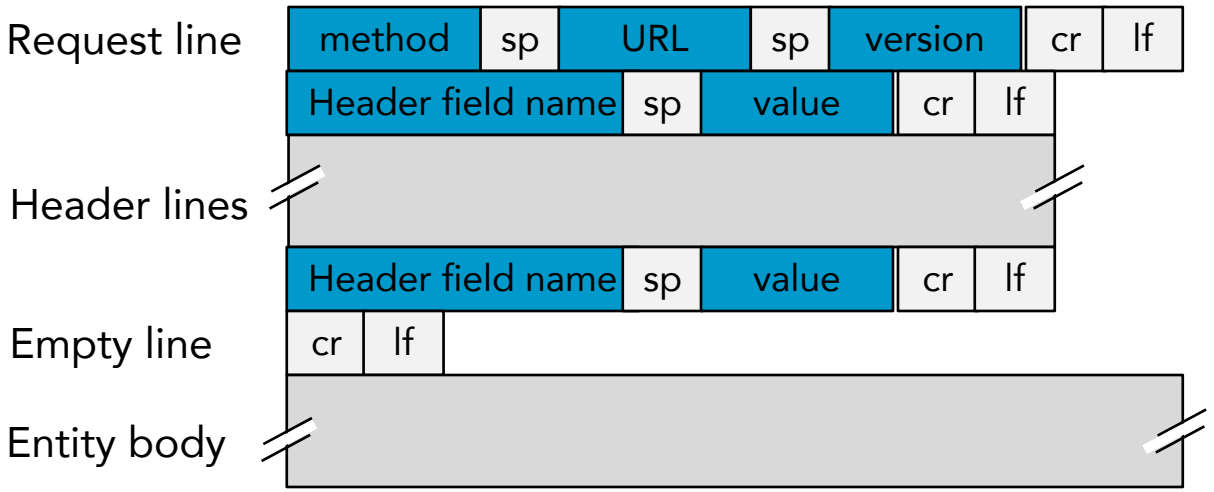
- A coarse estimate – 1 RTTs for first two parts of TCP 3-way handshake plus request (combined with third part) and response – 2 RTTs + transmission time at the server for the HTML file



# HTTP with Persistent Connections

- Issues with non-persistent connections
  - Need to established a new connection with each requested object, allocating TCP buffers, variables, …
  - Each objects pays 2 RTT for delivery
- HTTP 1.1 – Persistent connections
  - Server leaves the TCP connection open (configurable timeout of 10-15s)
  - Multiple web pages residing on the same server can be sent to the same client over it
  - Default mode, pipelining; HTTP/2 allows multiple requests interleaving and prioritization

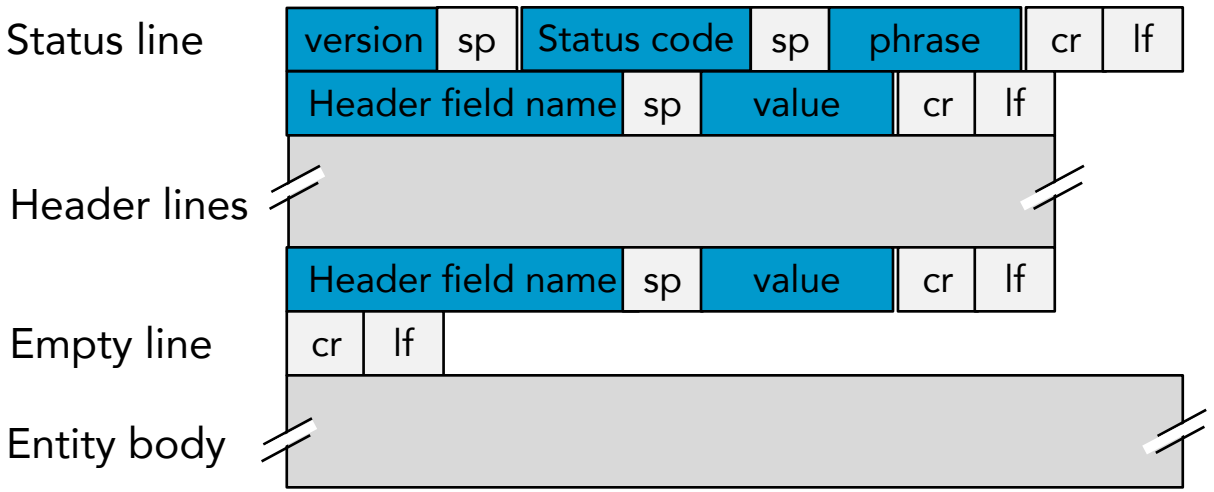
# HTTP Message Format – Requests




**GET**: to request a data
**POST**: to post data to the server, and perhaps get a response, too.
**PUT**: to create a new document on the server.
**DELETE**: to delete a document.
**HEAD**: like GET, but just return headers

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

# HTTP Message Format – Response




- **200 OK**: success
- **301 Moved Permanently**: redirects to another URL
- **403 Forbidden**: lack permission
- **404 Not Found**: URL is bad
- **500 Internal Server Error**

…

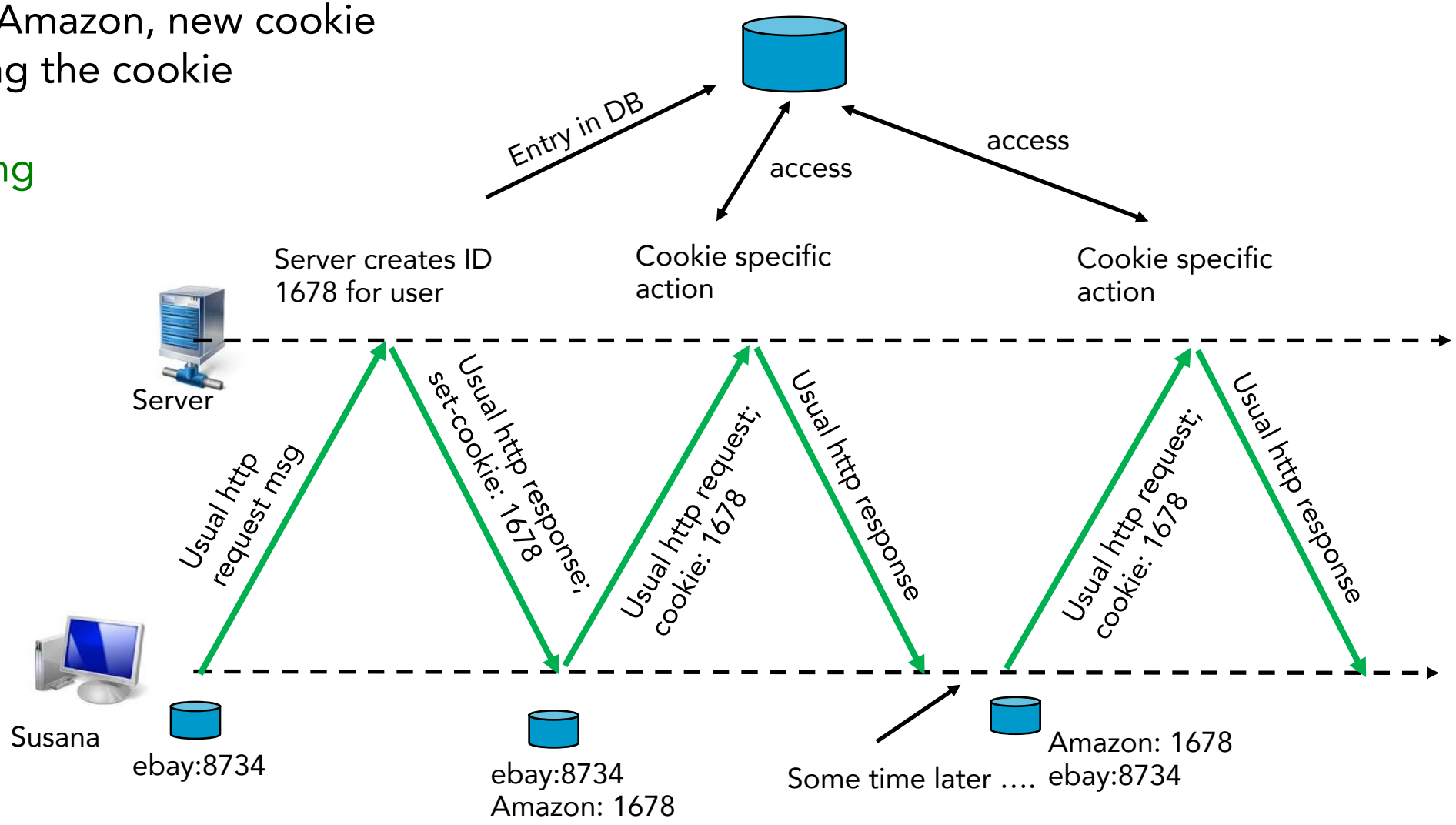
```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 17 Sep 2019 15:56:45 GMT
Server: Apache
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 221
Content-Type: text/html
... Data ...
```

# Cookies anyone?

- HTTP is stateless, easier for server design and higher scalability
- ***But*** sometimes you want to identify users across interactions
  - To restrict access, to serve specific content, … → cookies [RFC 6265]
- Four components
  - A cookie header line in the HTTP response
  - A cookie header line in the request
  - A cookie file kept on the user's end and managed by the user's browser
  - A back-end database at the web site

# Keeping user state with cookies

- First time visit to Amazon, new cookie
- Second time, bring the cookie along …
- One-click shopping
- Personalized ads
- Privacy concerns



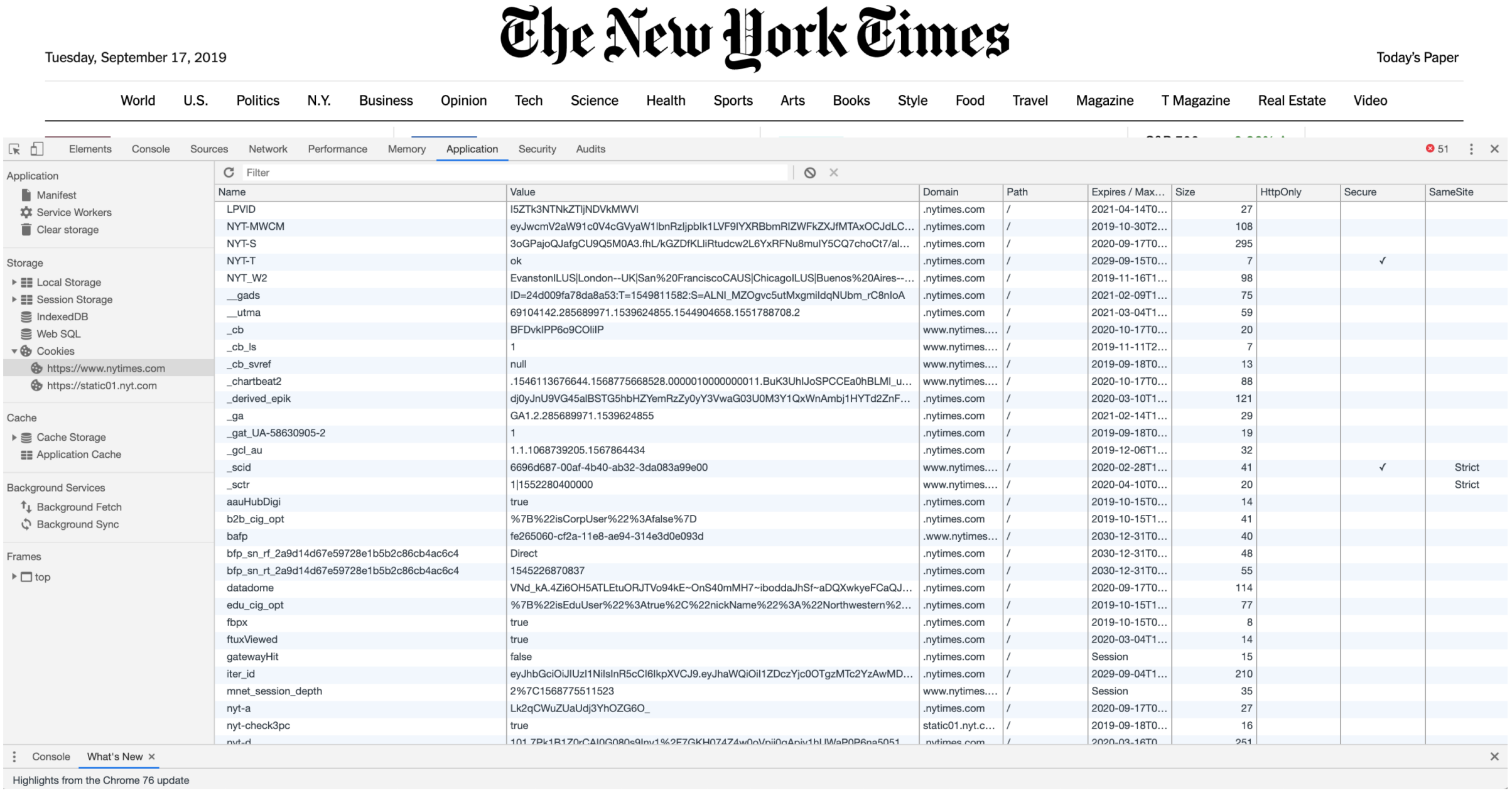
# Third party cookie, pixels, and tags

- Recall, that an HTTP response may include a cookie.
  - Cookies are random strings stored by your browser and included in every request to the same domain.
  - Cookies are a way for the browser to remind a website of your identity.
- Third party cookies are cookies from a domain different that the currently viewed web page.
  - Often enabled with a one-pixel GIF image included in the page:

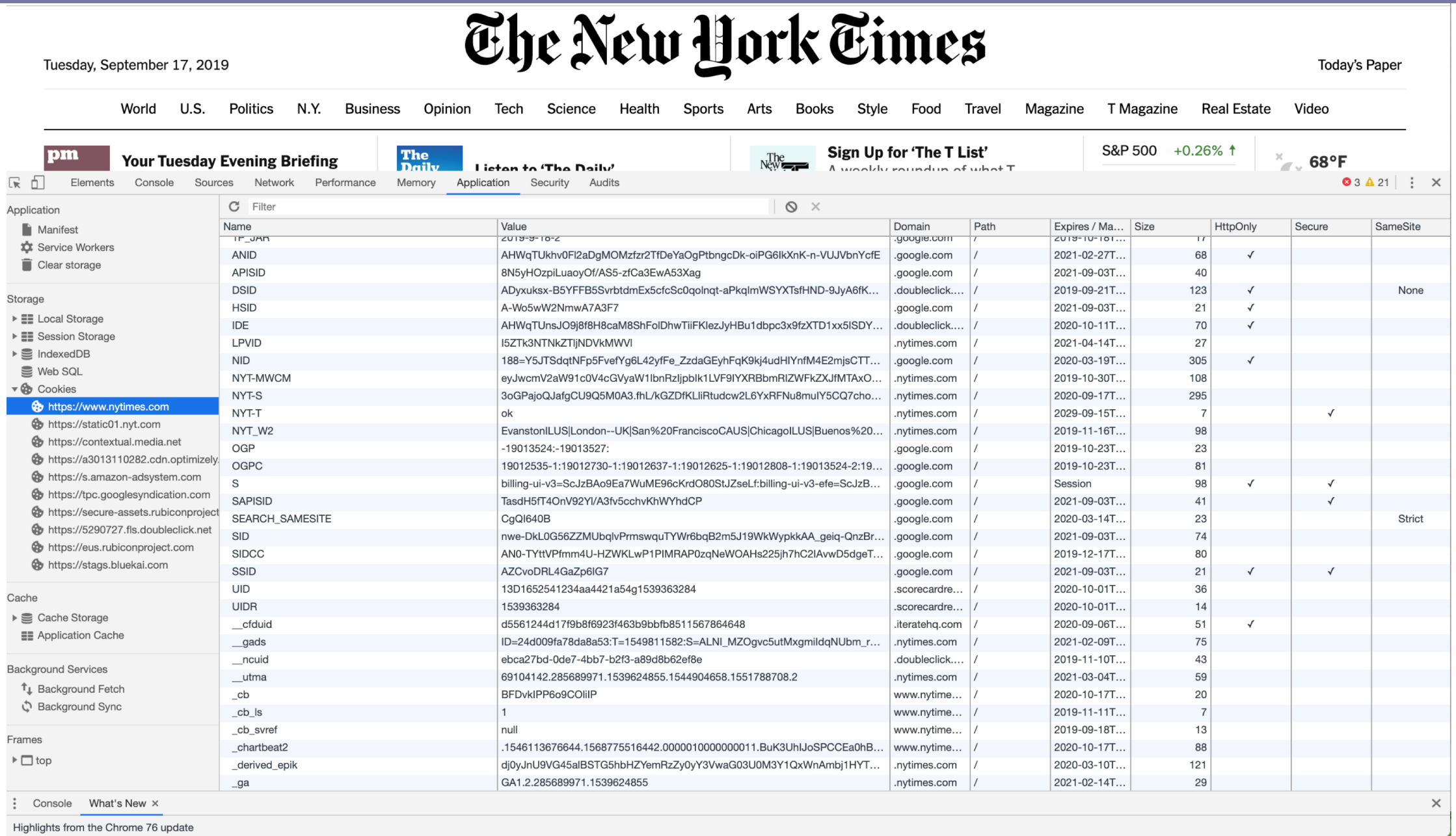
```
<img src="http://facebook.com/tracker/pix.gif">
```

  - Causes browser to send a request to facebook.com (including your Facebook cookie) even though I'm visiting a page unrelated to Facebook.
  - The request has a "Referer:" header listing the current URL.
  - Thus, Facebook (for example) learns about everything you do on the web.

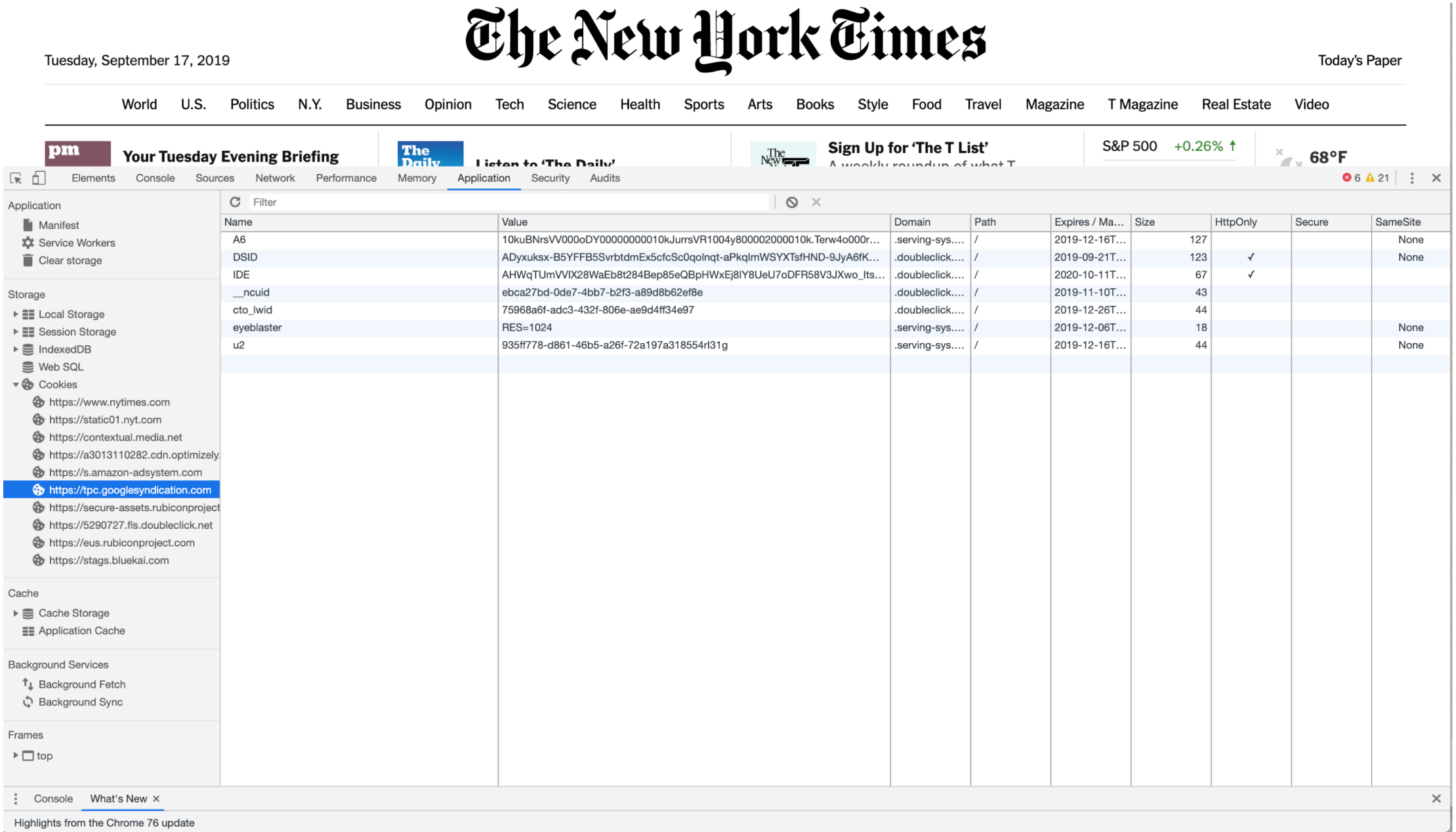
# Visiting Northwestern's webpage (with uBlock Origin)



# New York Times homepage (allowing all cookies)

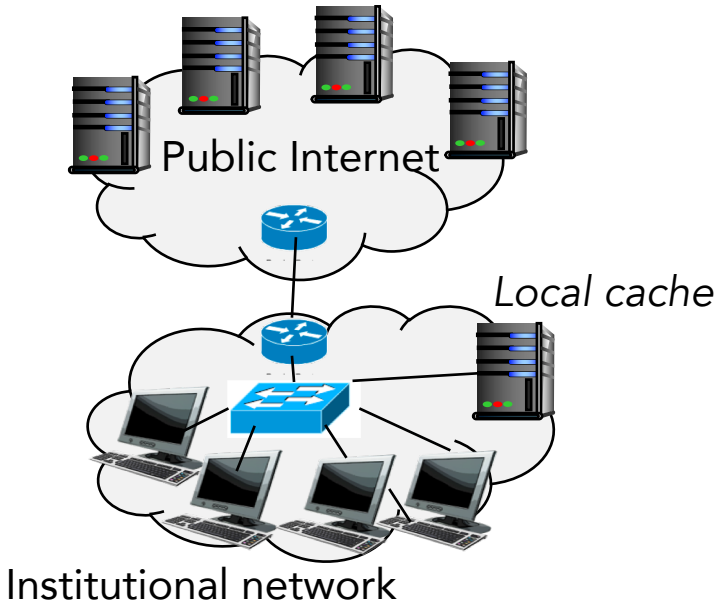


# New York Times homepage (allowing all cookies)



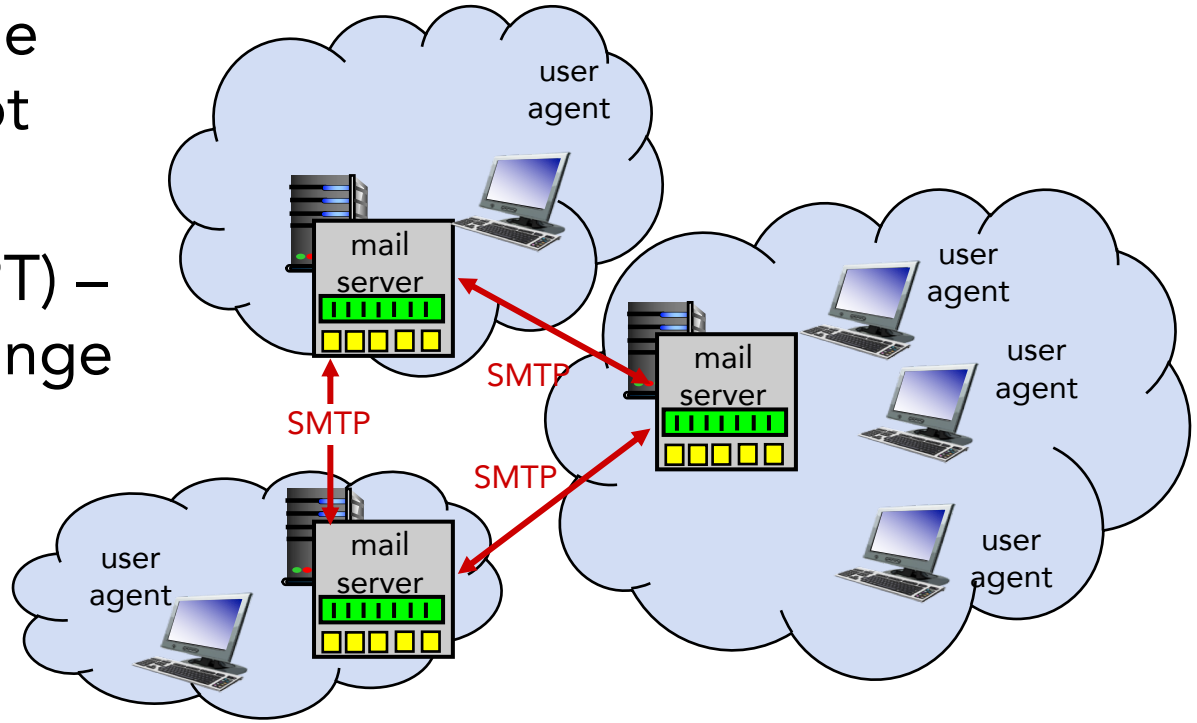
# Web caching

- Web cache, aka proxy server
  - Serves requests on behalf of an origin web server
  - Typically purchased/installed by an ISP
- Benefits
  - Can reduce response time for a client request, more so if the bottleneck bw client-to-server << clieny-to-proxy
  - Can reduce traffic on the access link → $$$
  - … Content Distribution Networks (CDNs)
- But what if the copy is stale?
  - Conditional GET "if-modified-since" header line
  - If not modified since, 304 response



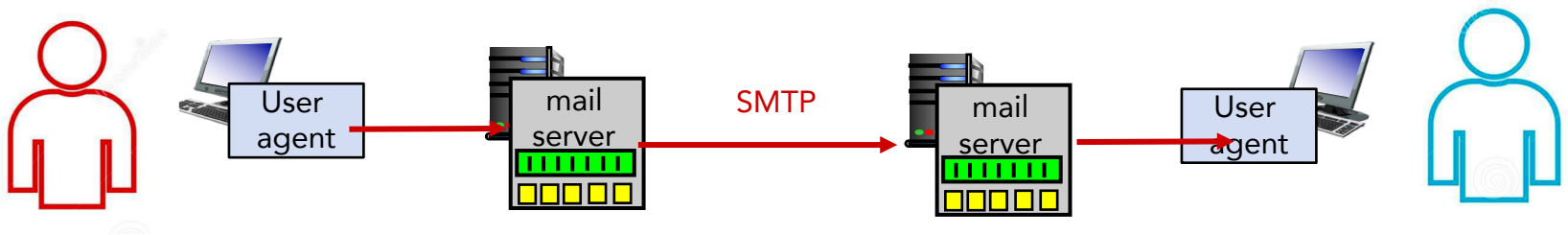
# Email and SMTP

- Internet first popular application
  - Like Snail mail, asynchronous
- Three major components
  - User agents – Let users read/reply/forward/… emails (e.g., Apple Mail)
  - Mail servers – Each user has a queue where emails sent/received are kept after sent/before delivery
  - Simple Mail Transfer Protocol (SMPT) – Application-layer protocol to exchange emails between mail servers



# Simple Mail Transport Protocol

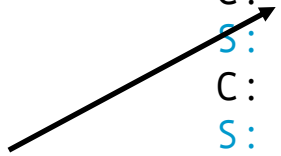
- Another protocol built on top of TCP (RFC 2821)
- Original RFC from 1982 but older than that
  - A bit archaic – body of 7 of all mail messages in 7-bit ASCII
  - Encoding/decoding of binary multimedia to ASCII before/after transfer
- Basic operation
  - Alice invokes user agent (UA) to send email to bob@someschool.edu
  - Alice's UA send msg to her email server, where it is put in the queue
  - Client's side of SMTP running on Alice's server sends msg over TCP
  - …
  - No intermediate mail servers



# Example

S: means server
C: means client

Commands



```
S: 220 smtp.example.com ESMTP Postfix
C: HELO relay.example.com
S: 250 smtp.example.com, I am glad to meet you
C: MAIL FROM:<bob@example.com>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: RCPT TO:<theboss@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Bob Example" bob@example.com
C: To: Alice Example alice@example.com
C: Cc: theboss@example.com
C: Date: Tue, 15 January 2008 16:02:43 -0500
C: Subject: Test message
C:
C: Hello Alice.
C: This is a test message with 5 header fields and 4 lines in the message body.
C: Your friend,
C: Bob
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
{The server closes the connection}
```

*Introduction*

*Sender's/recipient's emails*

*Body of email*

# Try SMTP for yourself

## It's one of the simplest protocols

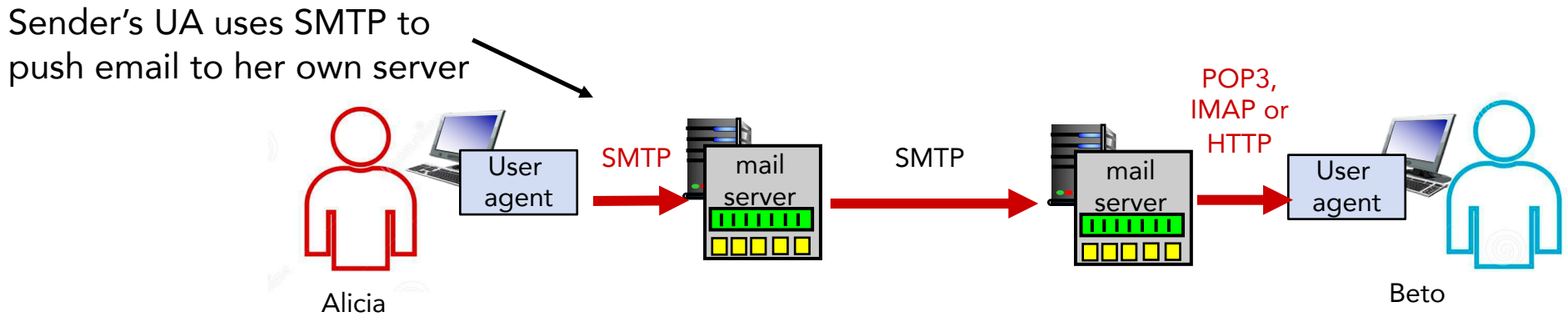
```
[fabianb@santos ~]$ nslookup -type=MX cs.northwestern.edu
Server:          129.105.5.98
Address: 129.105.5.98#53
cs.northwestern.edu       mail exchanger = 0 barra.eecs.northwestern.edu.
$ telnet barra.eecs.northwestern.edu 25
helo santos.cs.northwestern.edu
250 barra.eecs.northwestern.edu Hello santos.cs.northwestern.edu [129.105.44.79], pleased to meet
you
mail from: <fabianb@cs.northwestern.edu>
250 Sender <fabianb@cs.northwestern.edu> OK
rcpt to: <fabianb@cs.northwestern.edu>
250 Recipient <fabianb@cs.northwestern.edu> OK
DATA
354 Start mail input; end with <CRLF>.<CRLF>
this is a test
.
250 Ok: queued as 1519CBCA4C8
quit
Connection closed by foreign host.
```

## Compared with HTTP …

- Both use persistent TCP connections
- But SMTP is mainly a push protocol
- Each message, including the body needs to be in 7b ASCII format
- SMTP puts all parts of a message, images and txt, in one message

# Accessing email in the new millennium

- Up until the early 1990s, you would login onto a server host and execute a mail reader on that host
  - To check email on your laptop, you would need it on *all the time* just so you can send queued email and receive email at any time …
- Today's mail access uses a client-server architecture
  - Alicia's UA uses SMPT to push her email to her mail server …
    - That way her mail server can keep trying if Beto's mail server were not available
  - A different mail access protocol to transfer Beto's emails to his laptop



# Mail Access Protocols

- POP3 – Simple protocol (RFC 1939) with limited functionality
  - Access over TCP in 3 phases
    - Authorization – username/password exchange in clear text
    - Transaction – Retrieve messages, mark for deletion, get stats; download-and-delete (not great if you have more than one machine) or download-and-keep mode
    - Update – When quitting, carry on updates
- IMAP – More complex (RFC 3501), an improvement over POP3
  - Users can create folders in the server and move emails between them, search, get parts of a message (e.g., headers) if on a poor connection
- Web-based e-mail – Starting with Hotmail in mid 1990s, all exchanges with the mail server over HTTP
  - Beto to receive and Alicia to send (instead of SMTP)

# Summary

- We looked at concepts and implementation aspects of network applications …
- Earlier lectures provide a vague definition of a protocol – the format and order of the messages exchanged between communication entities, and the actions they take on the transmission/reception of messages or some other event …
- Our discussion made it a bit more concrete …
- But there's more to go with DNS and CDNs …