HTTP/2

Recent protocol changes and their impact on web application performance

Mark Friedman
Demand Technology Software
markf@demandtech.com
http://computerperformancebydesign.com

Presentation Outline

- Background
- Review the major changes adopted in HTTP/2 protocol
  - multiplexing
  - server push
  - Priority
- Performance impact of web site architecture
- Highlight areas where HTTP/2 and the default TCP congestion control policy may conflict

Background

- First change to the HTTP standard since 1999
  - HTTP/1.1 was a set of changes associated with session-oriented, web applications that deliver dynamic HTML web pages

- IETF HTTP Working Group recently adopted most, but not all, protocol changes proposed in a large scale Google experiment called SPDY
  - Designed to improve web application performance
  - HTTP/2 support in the web server and client will build on SPDY
  - SPDY benefits certain types of web sites more than others

"HTTP/2 isn't magic Web performance pixie dust; you can't drop it in and expect your page load times to decrease by 50%. It's more accurate to view the new protocol as removing some key impediments to performance; once browsers and servers learn how and when to take advantage of that, performance should start incrementally improving."

- **Mark Nottingham, chairperson of the IETF HTTP Working Group, from his blog, setting expectations for the transition to HTTP/2.**

Background to the HTTP/2 changes

- Google's SPDY experiment previews the most important of the changes to the HTTP standard
- Changes justified based on browser-based *Real User Measurements* (RUM) of web app performance
- Web site workload characterization:
  - HTTP/2 helps *monolithic* sites, but not necessarily *federated* web publishing

HTTP/2 introduces

- **Multiplexing**
- **Priority**
- **Server Push**
- **Header compression**
- **Improved performance with Transport Layer Security (compared to HTTPS)**

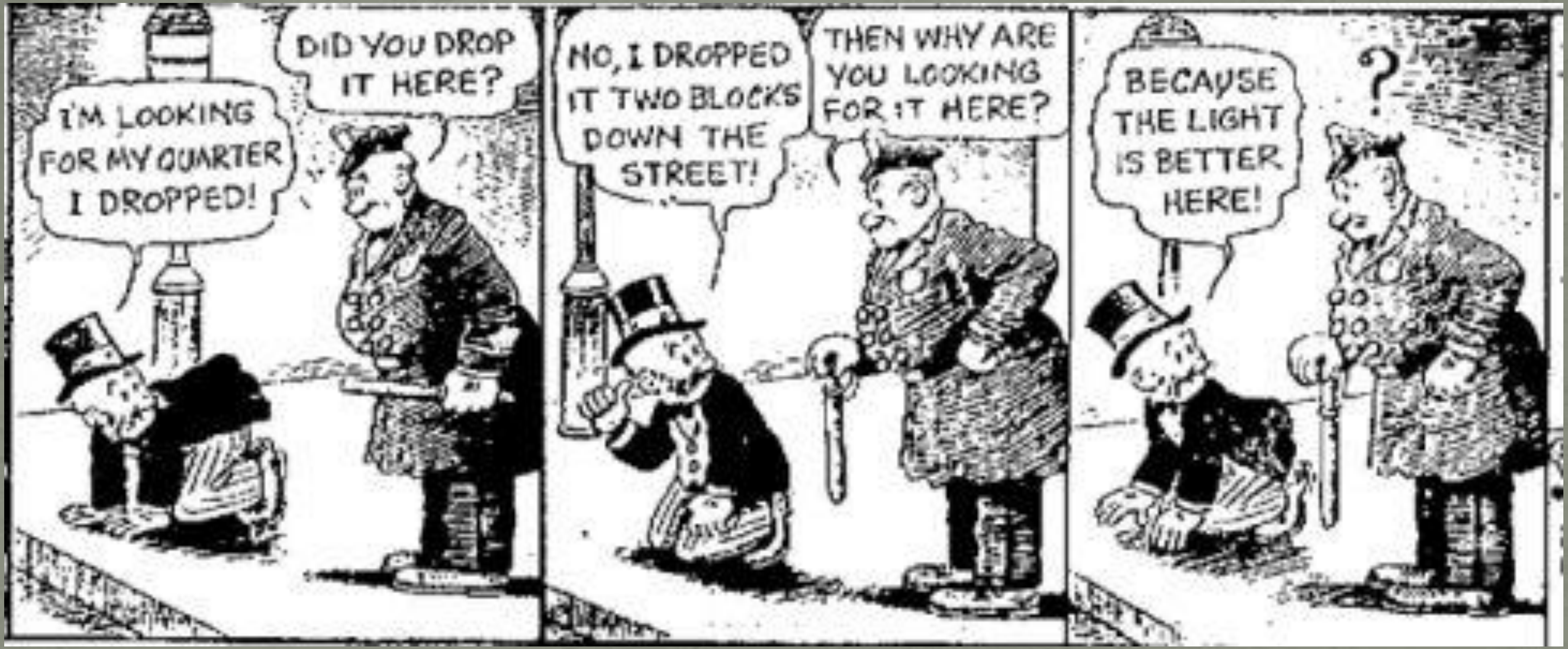
- **HTTP/2 requires *extensive* changes at both the web client and web server**

What HTTP/2 does not address

- JavaScript serialization delays
- Network-enabled applications that do not run inside the browser, but do rely on web services
  - e.g., native iPhone or Android apps
- TCP's use of Acknowledgements to confirm delivery of messages
- The TCP congestion control policy is unchanged
  - Consider adjusting some of the TCP defaults if your web site goes to HTTP/2

- Plus, HTTP/2 cannot repeal the laws of Physics that make network latency the fundamental source of web application performance problems

Using RUM measurements to justify the change suffers from the "Streetlight effect" (aka "Drunkard's Search")



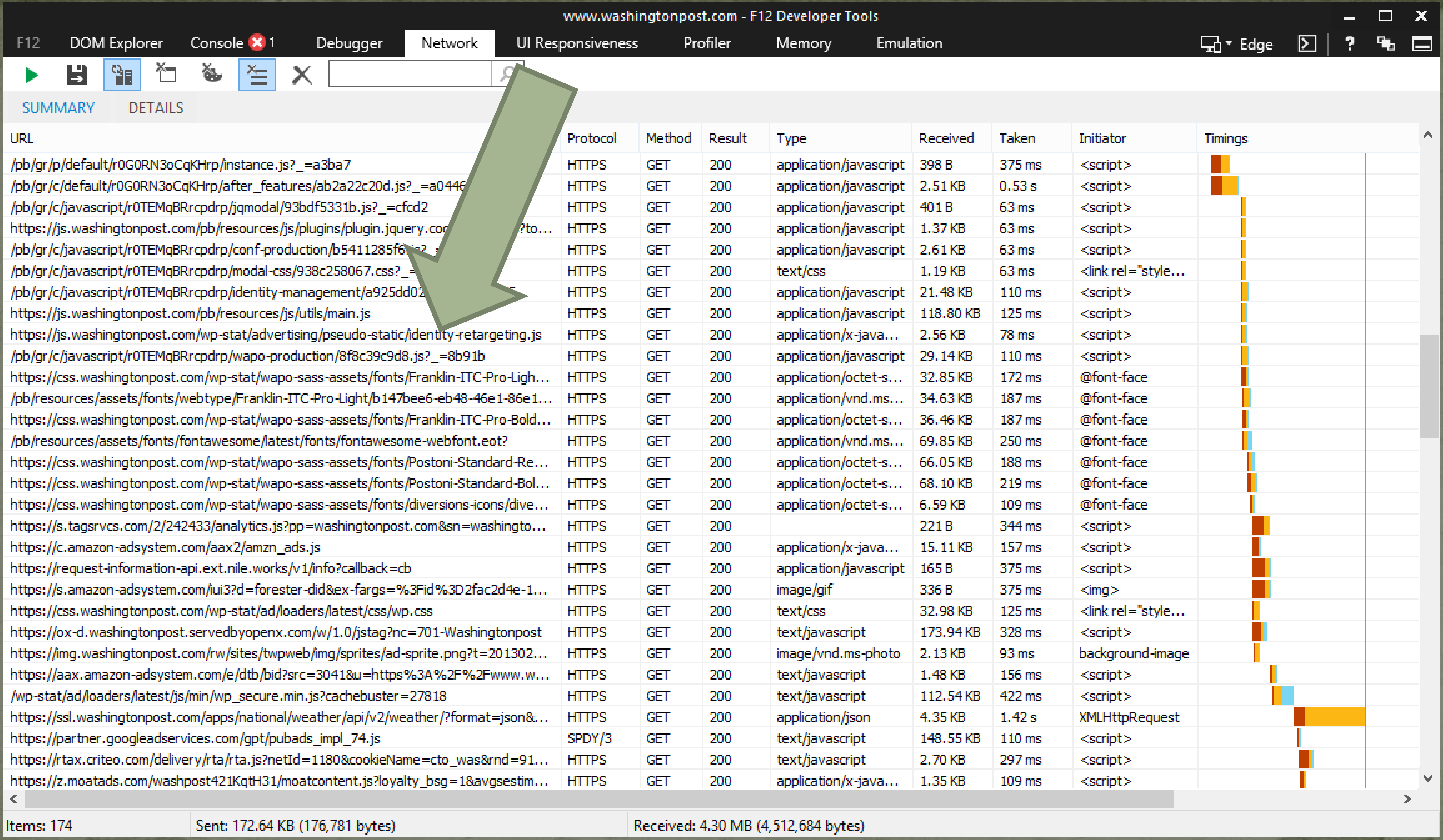
See **http://quoteinvestigator.com/2013/04/11/better-light/**.

"Streetlight effect" in computer performance

- Observational bias that favors the measurements we can readily acquire without sufficient regard for how valid and reliable those measurements are.
  - **Real User Measurements** (RUM) of web Page Load Time were used to validate and justify the HTTP/2 design decisions, despite their known limitations
  - Absent an understanding of the key characteristics of web application workloads that most impact performance

Key characterization of web application workloads that most impact performance under HTTP/2

- monolithic web publishing utilizes a very small number of domains
- federated web publishing where content may be pulled from as mainly as 50 affiliated domains



URL	Protocol	Method	Result	Type	Received	Taken	Initiator
/pb/gr/p/default/r0G0RN3oCqKHrp/instance.js?_=a3ba7	HTTPS	GET	200	application/javascript	398 B	375 ms	<script>
/pb/gr/c/default/r0G0RN3oCqKHrp/after_features/ab2a22c20d.js?_=a0446	HTTPS	GET	200	application/javascript	2.51 KB	0.53 s	<script>
/pb/gr/c/javascript/r0TEMqBRrcpdrp/jqmodal/93bdf5331b.js?_=cfcd2	HTTPS	GET	200	application/javascript	401 B	63 ms	<script>
https://js.washingtonpost.com/pb/resources/js/plugins/plugin.jquery.coo[illegible]?to…	HTTPS	GET	200	application/javascript	1.37 KB	63 ms	<script>
/pb/gr/c/javascript/r0TEMqBRrcpdrp/conf-production/b5411285f6[illegible]	HTTPS	GET	200	application/javascript	2.61 KB	63 ms	<script>
/pb/gr/c/javascript/r0TEMqBRrcpdrp/modal-css/938c258067.css?_=[illegible]	HTTPS	GET	200	text/css	1.19 KB	63 ms	<link rel="style…
/pb/gr/c/javascript/r0TEMqBRrcpdrp/identity-management/a925dd02[illegible]	HTTPS	GET	200	application/javascript	21.48 KB	110 ms	<script>
https://js.washingtonpost.com/pb/resources/js/utils/main.js	HTTPS	GET	200	application/javascript	118.80 KB	125 ms	<script>
https://js.washingtonpost.com/wp-stat/advertising/pseudo-static/identity-retargeting.js	HTTPS	GET	200	application/x-java…	2.56 KB	78 ms	<script>
/pb/gr/c/javascript/r0TEMqBRrcpdrp/wapo-production/8f8c39c9d8.js?_=8b91b	HTTPS	GET	200	application/javascript	29.14 KB	110 ms	<script>
https://css.washingtonpost.com/wp-stat/wapo-sass-assets/fonts/Franklin-ITC-Pro-Ligh…	HTTPS	GET	200	application/octet-s…	32.85 KB	172 ms	@font-face
/pb/resources/assets/fonts/webtype/Franklin-ITC-Pro-Light/b147bee6-eb48-46e1-86e1…	HTTPS	GET	200	application/vnd.ms…	34.63 KB	187 ms	@font-face
https://css.washingtonpost.com/wp-stat/wapo-sass-assets/fonts/Franklin-ITC-Pro-Bold…	HTTPS	GET	200	application/octet-s…	36.46 KB	187 ms	@font-face
/pb/resources/assets/fonts/fontawesome/latest/fonts/fontawesome-webfont.eot?	HTTPS	GET	200	application/vnd.ms…	69.85 KB	250 ms	@font-face
https://css.washingtonpost.com/wp-stat/wapo-sass-assets/fonts/Postoni-Standard-Re…	HTTPS	GET	200	application/octet-s…	66.05 KB	188 ms	@font-face
https://css.washingtonpost.com/wp-stat/wapo-sass-assets/fonts/Postoni-Standard-Bol…	HTTPS	GET	200	application/octet-s…	68.10 KB	219 ms	@font-face
https://css.washingtonpost.com/wp-stat/wapo-sass-assets/fonts/diversions-icons/dive…	HTTPS	GET	200	application/octet-s…	6.59 KB	109 ms	@font-face
https://s.tagsrvcs.com/2/242433/analytics.js?pp=washingtonpost.com&sn=washingto…	HTTPS	GET	200		221 B	344 ms	<script>
https://c.amazon-adsystem.com/aax2/amzn_ads.js	HTTPS	GET	200	application/x-java…	15.11 KB	157 ms	<script>
https://request-information-api.ext.nile.works/v1/info?callback=cb	HTTPS	GET	200	application/javascript	165 B	375 ms	<script>
https://s.amazon-adsystem.com/iui3?d=forester-did&ex-fargs=%3Fid%3D2fac2d4e-1…	HTTPS	GET	200	image/gif	336 B	375 ms	<img>
https://css.washingtonpost.com/wp-stat/ad/loaders/latest/css/wp.css	HTTPS	GET	200	text/css	32.98 KB	125 ms	<link rel="style…
https://ox-d.washingtonpost.servedbyopenx.com/w/1.0/jstag?nc=701-Washingtonpost	HTTPS	GET	200	text/javascript	173.94 KB	328 ms	<script>
https://img.washingtonpost.com/rw/sites/twpweb/img/sprites/ad-sprite.png?t=201302…	HTTPS	GET	200	image/vnd.ms-photo	2.13 KB	93 ms	background-image
https://aax.amazon-adsystem.com/e/dtb/bid?src=3041&u=https%3A%2F%2Fwww.w…	HTTPS	GET	200	text/javascript	1.48 KB	156 ms	<script>
/wp-stat/ad/loaders/latest/js/min/wp_secure.min.js?cachebuster=27818	HTTPS	GET	200	text/javascript	112.54 KB	422 ms	<script>
https://ssl.washingtonpost.com/apps/national/weather/api/v2/weather/?format=json&…	HTTPS	GET	200	application/json	4.35 KB	1.42 s	XMLHttpRequest
https://partner.googleadservices.com/gpt/pubads_impl_74.js	SPDY/3	GET	200	text/javascript	148.55 KB	110 ms	<script>
https://rtax.criteo.com/delivery/rta/rta.js?netId=1180&cookieName=cto_was&rnd=91…	HTTPS	GET	200	text/javascript	2.70 KB	297 ms	<script>
https://z.moatads.com/washpost421KqtH31/moatcontent.js?loyalty_bsg=1&avgsestim…	HTTPS	GET	200	application/x-java…	1.35 KB	109 ms	<script>

Items: 174 | Sent: 172.64 KB (176,781 bytes) | Received: 4.30 MB (4,512,684 bytes)

Federated web sites

- Building the page requires access > 10 distinct domains
  - Among the Top 500 web sites, some pull together content > 50 domains
  - e.g., Requests to 3rd party Ad servers
- Some web publishing sites were federated deliberately to take advantage of the web client's support for concurrent TCP sessions
  - Improved throughput because concurrent TCP sessions allow content from the same domain to be downloaded in parallel
  - Whenever the domains are co-located, this practice is known as ***sharding***
    - handshaking protocol required to establish each individual TCP session, so domain sharding has to be done carefully

SPDY

- Predecessor of HTTP/2 multiplexing
- Developed at Google
- Implemented on Chrome and across the Google web properties
  - Developers report a 15% overall improvement in Page Load Times with SPDY
    - Fewer TCP connections
    - Smaller GET Requests
    - number of packets shows 20% reduction
    - Google Search page shows minimal improvement (already highly optimized)
      - < 20 GET Requests (most of which are cached on the client)
      - < 5 domains
    - But looks promising for bandwidth thirsty sites like YouTube
  - SPDY white paper reports 50% reduction in page load times

SPDY criticism

- Guy Podjarney, a CTO at Akamai blogs "not as SPDY as you thought"
- He reports,
  - "SPDY, on average, is only about 4.5% faster than plain HTTPS, and is in fact about 3.4% slower than unencrypted HTTP"
  - SPDY improves performance under two sets of circumstances:
    - monolithic sites that consolidated content on a small number of domains
    - pages that did not block significantly during resolution of JavaScript files and .css style sheets
  - SPDY particularly benefits page composition for
    1. complex web pages,
    2. composed from Requests mainly directed to a single domain,
    3. where multiplexing is able to re-use a single TCP connection effectively

Evaluating SPDY

- Is SPDY a worthwhile improvement or is it just making the public Internet safer for cat videos (in HD, no less)?
  - e.g., https://youtu.be/UIrEM_9qvZU with 16M views
- Overall, web Page size and complexity are increasing, however

Year	Average web page size
2011	0.7 MB
2015	> 2 MB

- TCP Port number constrained to 16-bits, an upper limit on the number of concurrent sessions, so any relief is welcome

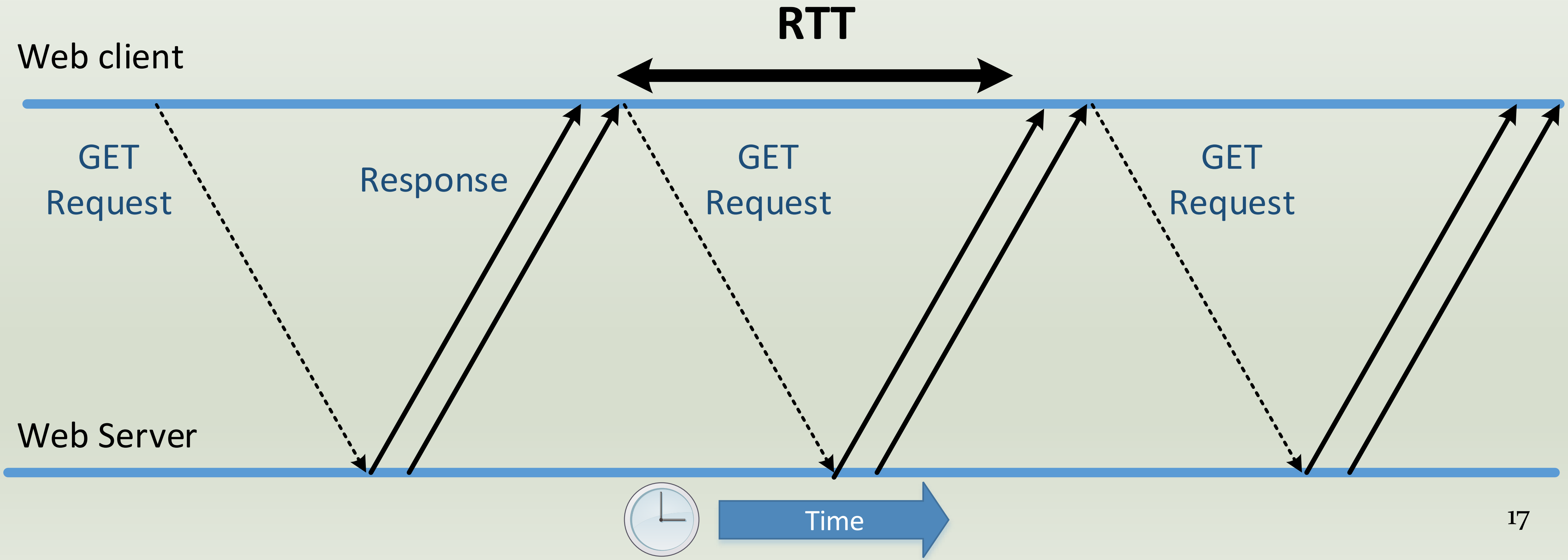
Major new features in HTTP/2:

- Multiplexing
- Priority
- Server Push
- Header compression
- Improved performance with Transport Layer Security (compared to HTTPS)

- HTTP/2 requires changes at both the web client and web server

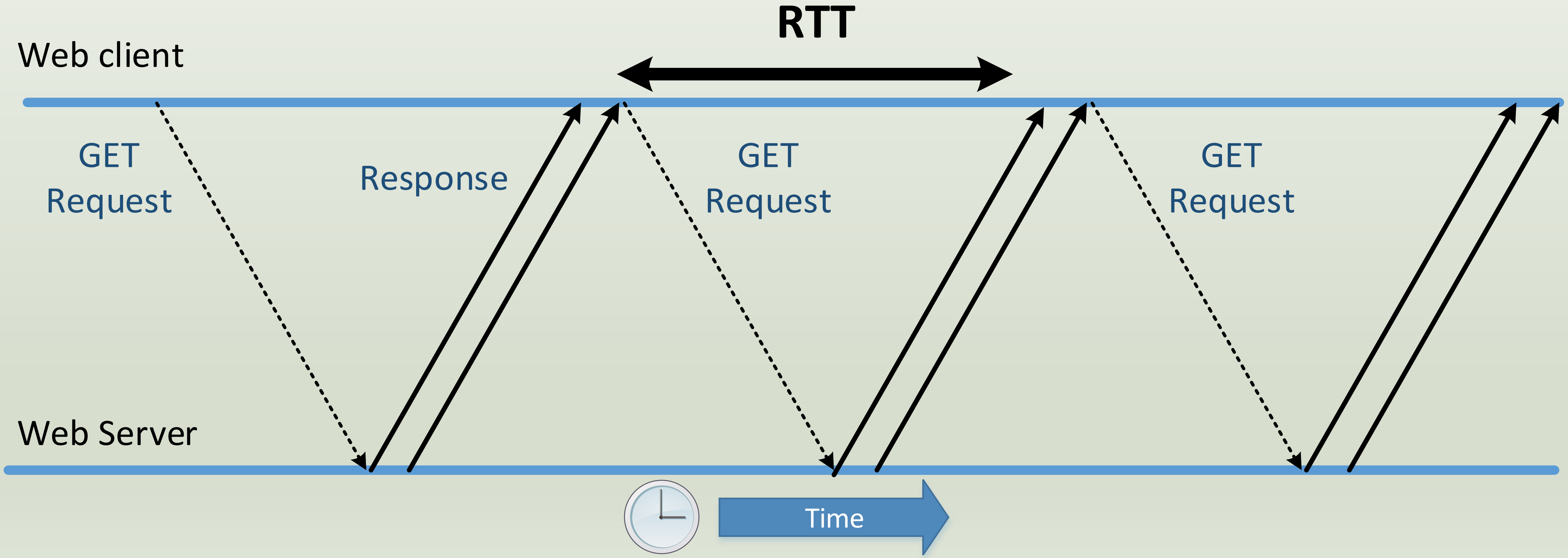
Page composition in HTTP/1.x

- Web client sends GET Requests to a web server **serially** over a single TCP connection.



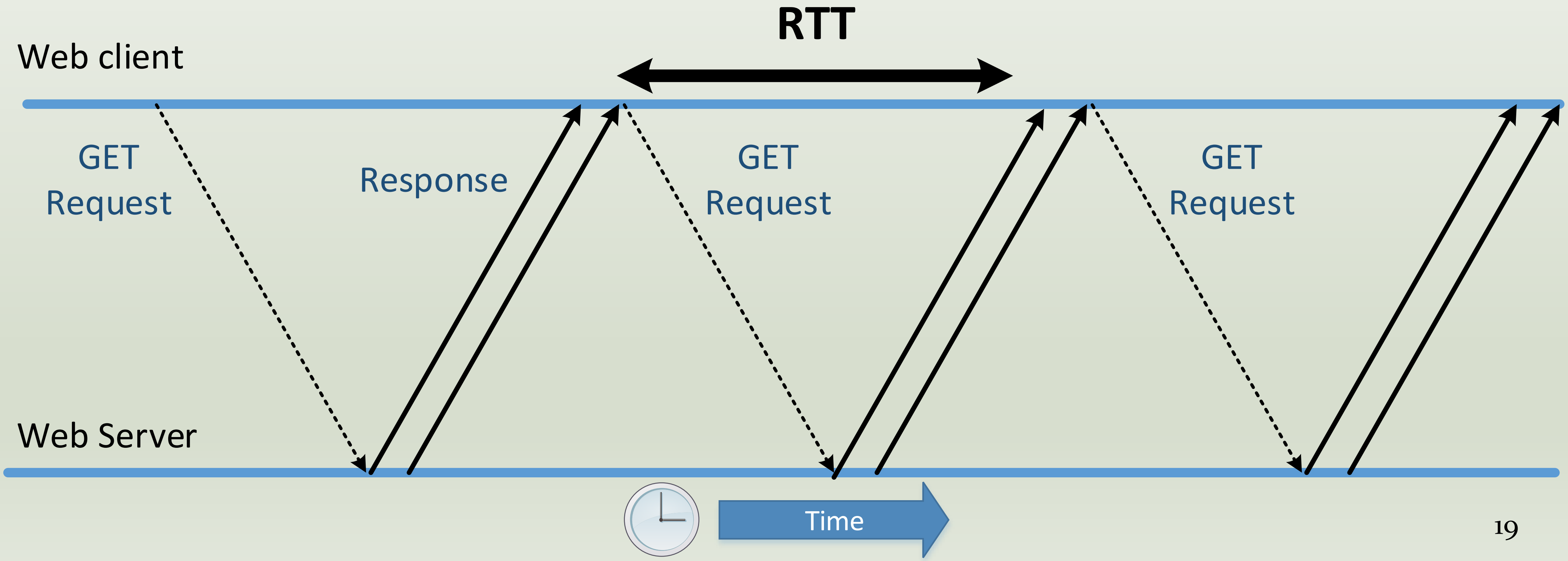
Page composition in HTTP/1.x

- Any follow-up GET Requests are **delayed** until the Response message from the previous Request is received.
- This delay is the **Round Trip Time** (RTT)



Round Trip Time (RTT)

- RTT = 2 * Network latency
- RTT affects Time To First Byte; bandwidth and HTTP object size affect Page Load Time



YSlow scalability model

- **Web page composition (usually) requires multiple GET Requests**

Assuming rendering time inside the web client is minimal,

Web Page Load Time = Render Time ≈ RoundTrips * RTT

where

$$\text{RoundTrips} = \sum_{i=1}^{n} \frac{httpObjectSize_i}{packetsize}$$

Parallelism in HTTP/1.1 rendering

- **Web servers are clustered using virtual IP addressing**
  - Sessionless (aka **REST**) Requests can be handled by any web server in the cluster
- **Multiple domains can be accessed concurrently**
  - Benefits federated sites
  - Benefits sharded sites
- **Multiple sessions can be established for each domain**
  - Diminishing returns expected from multiple sessions
- **Web services can be accessed asynchronously**

- **However, there is no explicit support for *multithreading* at the application level for JavaScript running on the browser**
  - JavaScript files must be downloaded and executed serially

A *better* YSlow scalability model

Assuming rendering time inside the web client is minimal,

Web Page Load Time = Render Time ≈ RoundTrips * RTT

where

$$\text{RoundTrips} = \sum_{i=1}^{n} \frac{httpObjectSize_i}{packetsize}$$

- **A degree of *parallelism* is obtained due to multiple sessions and multiple domains**
- **RTT is apt to *vary* by location/domain**

Key characterization of web application workloads that affect performance under HTTP/2

1. The number of distinct domains
2. the number of GET Requests directed to each domain
3. the distribution of the size of those objects

- monolithic web publishing utilizes a concise number of domains
- federated web publishing where content may be pulled from as mainly as 50 affiliated domains

Parallelism in web page composition in HTTP/1.x

- To improve performance, the web browser in HTTP/1.1 downloads individual content files in parallel

- Client can access multiple domains in parallel
  - Dynamic and static content is often split across separate web servers
    - Whenever these dmains are co-located, this is known as **domain sharding**
  - Static content is often cached on a CDN or in-house "edge" network
- Client can open multiple sessions to each web server domain
  - The official guideline is up six sessions per domain, but mileage varies with the browser and the platform

Parallelism in web page composition in HTTP/1.x

- To improve performance, the web browser in HTTP/1.1 downloads individual content files in parallel

- Effective when the sessions are relatively long-lived.
  - Each new domain may require a DNS Lookup
  - Handshaking for each new TCP Session requires 1 * RTT
  - Handshaking for HTTPS requires an additional RTT

- This parallelism works under HTTP/1.x because the HTTP protocol was originally designed to be *sessionless* and *connectionless*

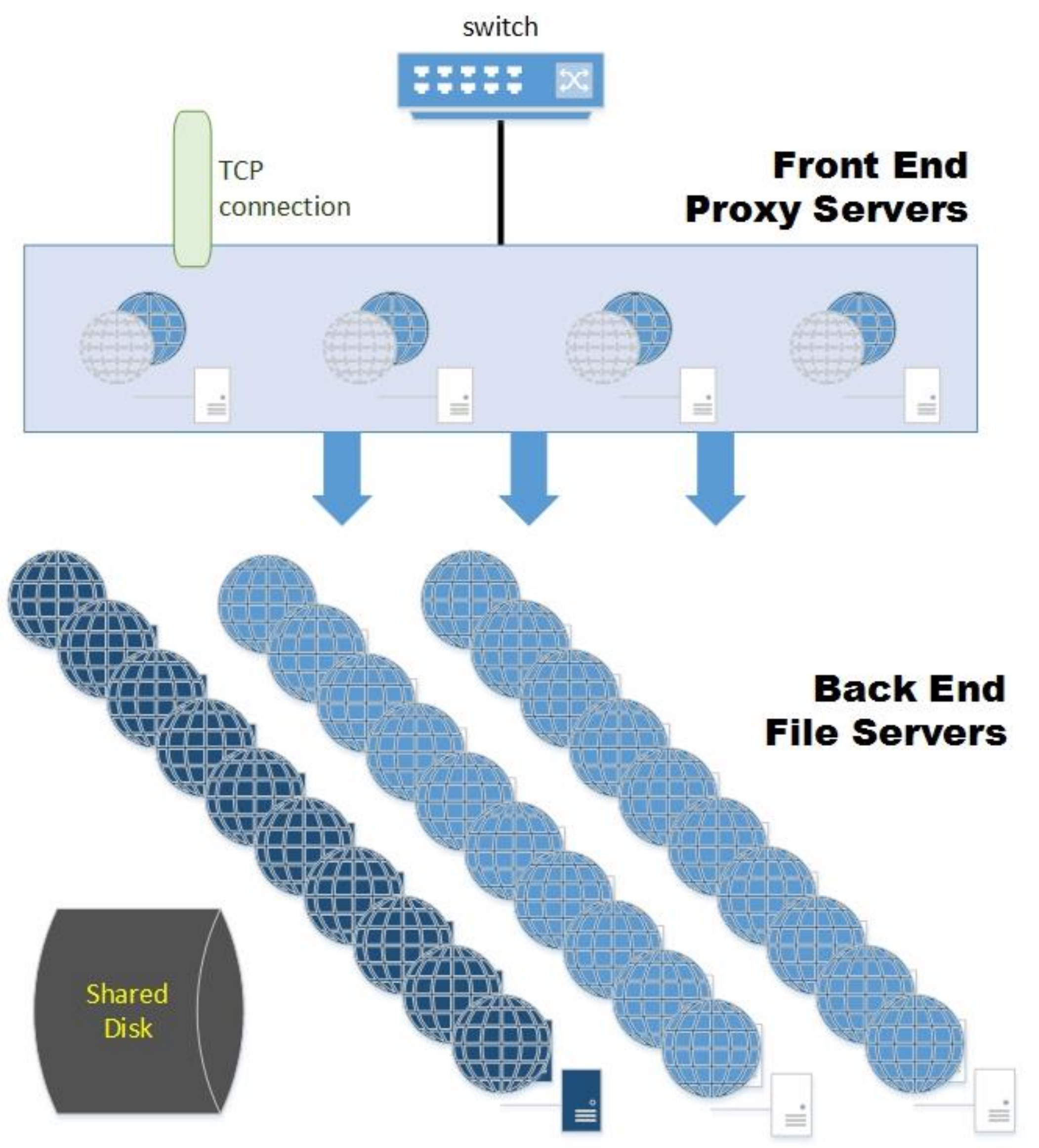
  - Any web server in the cluster can respond to any HTTP Request

  - HTTP sits atop TCP, which is session-oriented, which many web applications do exploit (e.g., session-aware ASP.NET apps on the Microsoft platform)

Parallelism in the web server infrastructure

- ***Any web server in the cluster can respond to any HTTP Request***

- Provisioned using
  - Virtualization
  - CDNs
  - Cloud (e.g., AWS)

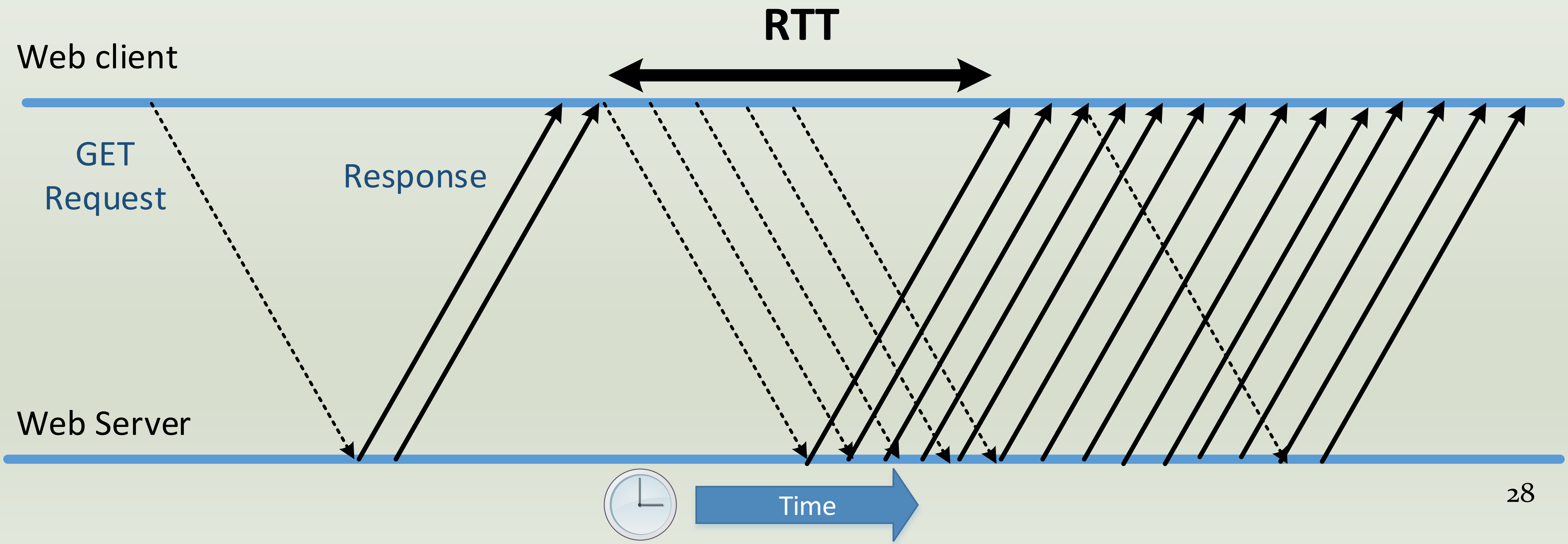


Parallelism in web page composition in HTTP/1.x

- Add explicit parallelism to the page using JavaScript to make asynchronous **XMLHttpRequests** to web services after the page is Loaded (aka, AJAX) and is (ostensibly) Ready for user input
  - A Best Practice for accessing 3rd party Advertising services, for example.
- Note: The web client downloads JavaScript and executes it **serially**
  - This is the reason why experts recommend placing all external JavaScript hrefs near the end of the HTML message

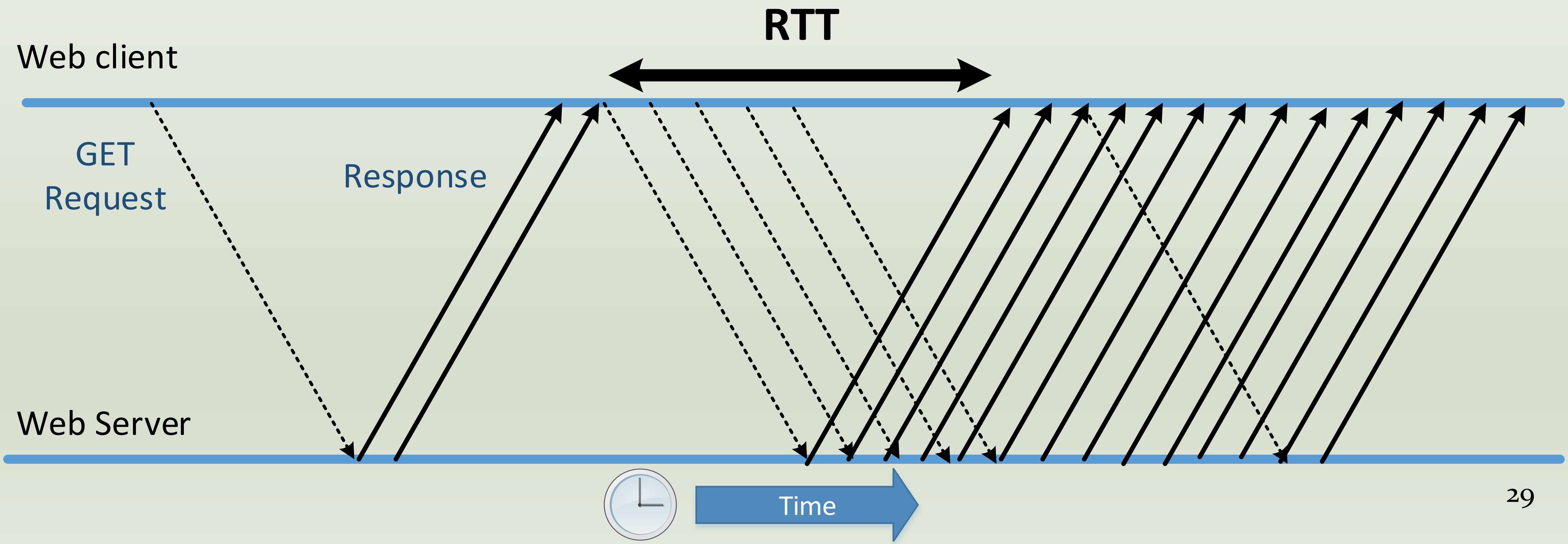
Multiplexing in HTTP/2

- Web browser can send multiple GET Requests without waiting for each individual Response



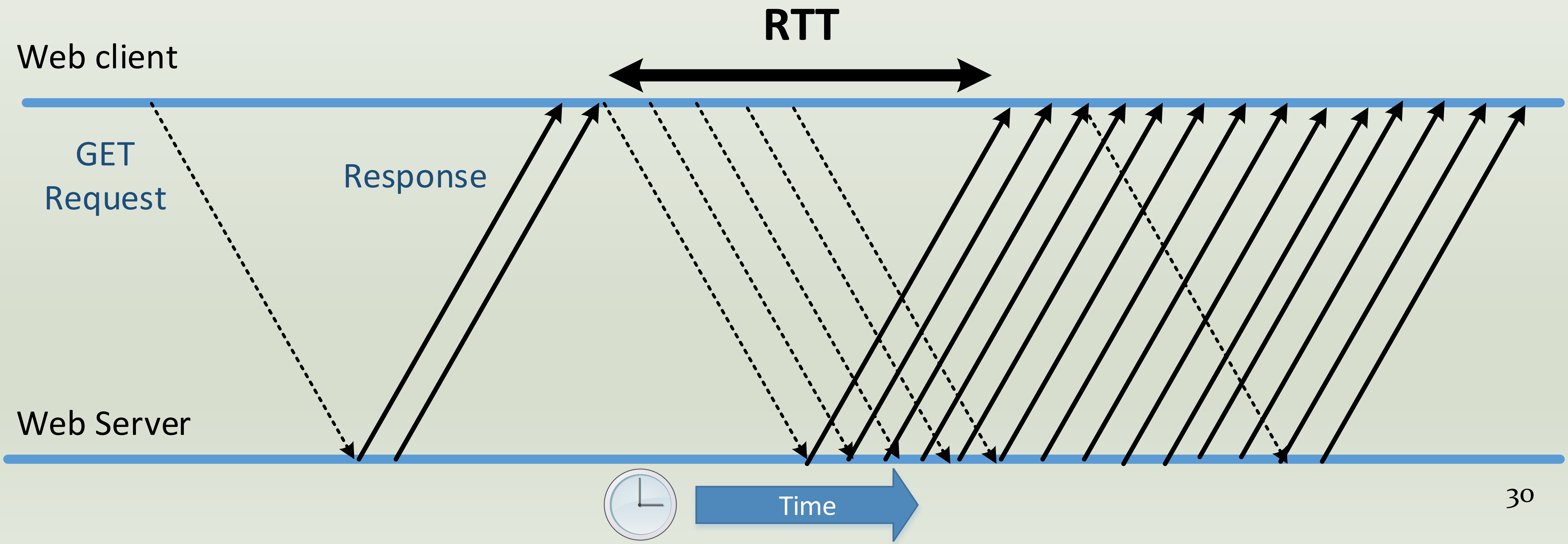
Multiplexing in HTTP/2

- Web server can send Response messages in any sequence
- Segments from multiple Response messages can be interleaved



Multiplexing in HTTP/2

- Achieve the same or higher levels of concurrency as HTTP/1.1 ***over a single TCP connection***



An Example: https:\\facebook.com

- Compare HTTP/1.1 to SPDY/3 access using Internet Explorer (IE 11)

1. DNS Lookup
2. HTTPS handshaking
   - SPDY exchanges two fewer packets to establish the secure connection
3. GET Request to www.facbook.com
   - Very large amount of cookie data is transmitted (> 1 packet)
4. FB server-side php builds an initial, custom Response message
   - ~ 550 KB
   - requires 2 seconds to transmit
   - contains a large number of external references: scripts, styles sheets, image files, video, and advertising content

An Example: https:\\facebook.com

- Comparing HTTP/1.1 to SPDY/3 multiplexing
  - Steps 1-4: SPDY = HTTP/1.1

5. Loading the full page then requires
   - 216 GET Requests and Response message sequences
   - transfers 7.24 MB of data over the wire
   - 3.6 seconds until Page Load event fires

6. JavaScript issuing XmlHttpRequests in the background continues to execute for ~20 seconds more

An Example: https:\\facebook.com

- Compare HTTP/1.1 to SPDY/3 access using Internet Explorer (IE 11)
  - Step 5: SPDY ≠ HTTP/1.1.
  - For example:
    - Early in the original Response message, 5 external .css files are referenced:

```
<link type="text/css" rel="stylesheet" href="https://fbstatic-a.akamaihd.net/rsrc.php/v2/yB/r/PQzGy_gthig.css" />
<link type="text/css" rel="stylesheet" href="https://fbstatic-a.akamaihd.net/rsrc.php/v2/yJ/r/cuqNSNZ2dlI.css" />
<link type="text/css" rel="stylesheet" href="https://fbstatic-a.akamaihd.net/rsrc.php/v2/yi/r/RH3rvDA7dSR.css" />
<link type="text/css" rel="stylesheet" href="https://fbstatic-a.akamaihd.net/rsrc.php/v2/yf/r/QFcEQNF3244.css" />
<link type="text/css" rel="stylesheet" href="https://fbstatic-a.akamaihd.net/rsrc.php/v2/yD/r/flQGK0biLk6.css" />
```

  - Residing on a Facebook web site affiliate devoted to static content:

```
href="https://fbstatic-a.akamaihd.net/
```

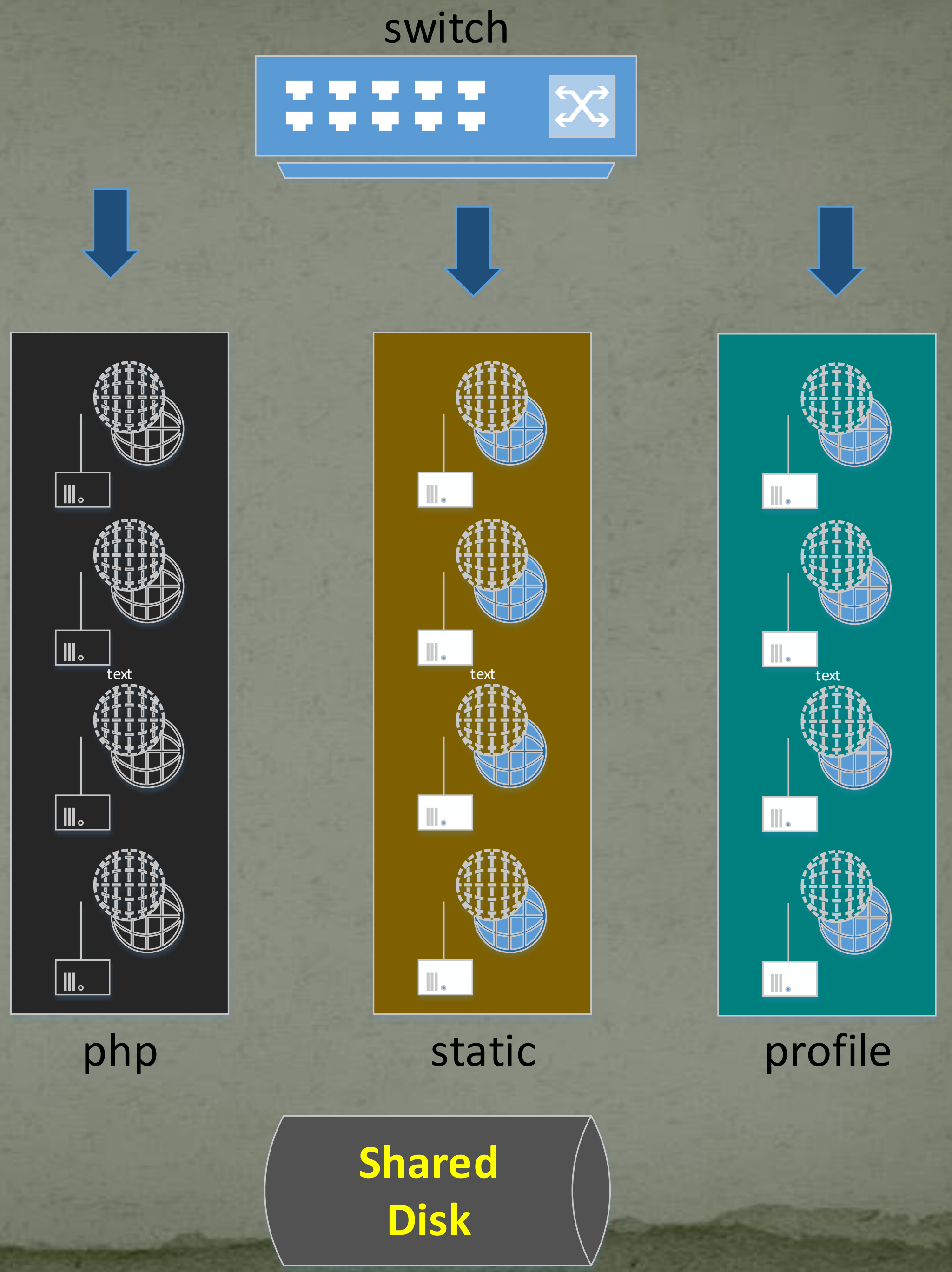
An Example: https:\\facebook.com

- Comparing HTTP/1.1 to SPDY/3 multiplexing
  - Steps 1-4: SPDY = HTTP/1.1

- **216 GET Requests**
- **But ¾ of the Requests are directed to just two web sites**
  - *fbstatic* domain where common style sheets, image files, and scripts are located
  - an *fbcdn-profile* domain where content specific to my Facebook profile and set of Friends was stored.

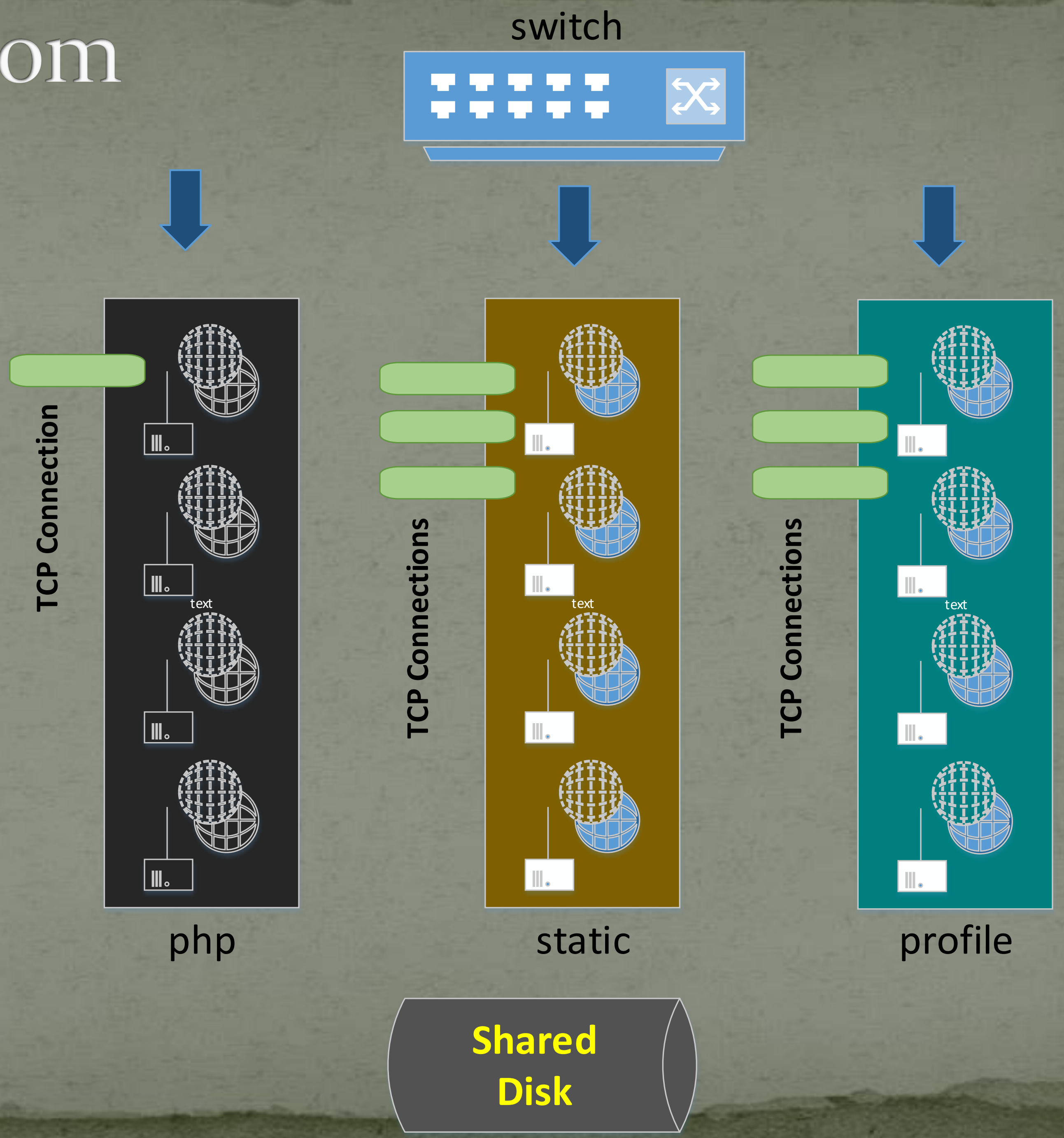
https:\\facebook.com

- **Clustered & Partitioned**
  - Front-end proxy/switch
  - PHP web servers
  - Back-end file servers
    - static content
    - profile content
  - Persistent back-store
- **Massively parallel**
  - Any web server in the cluster can respond to any *connectionless* HTTP Request



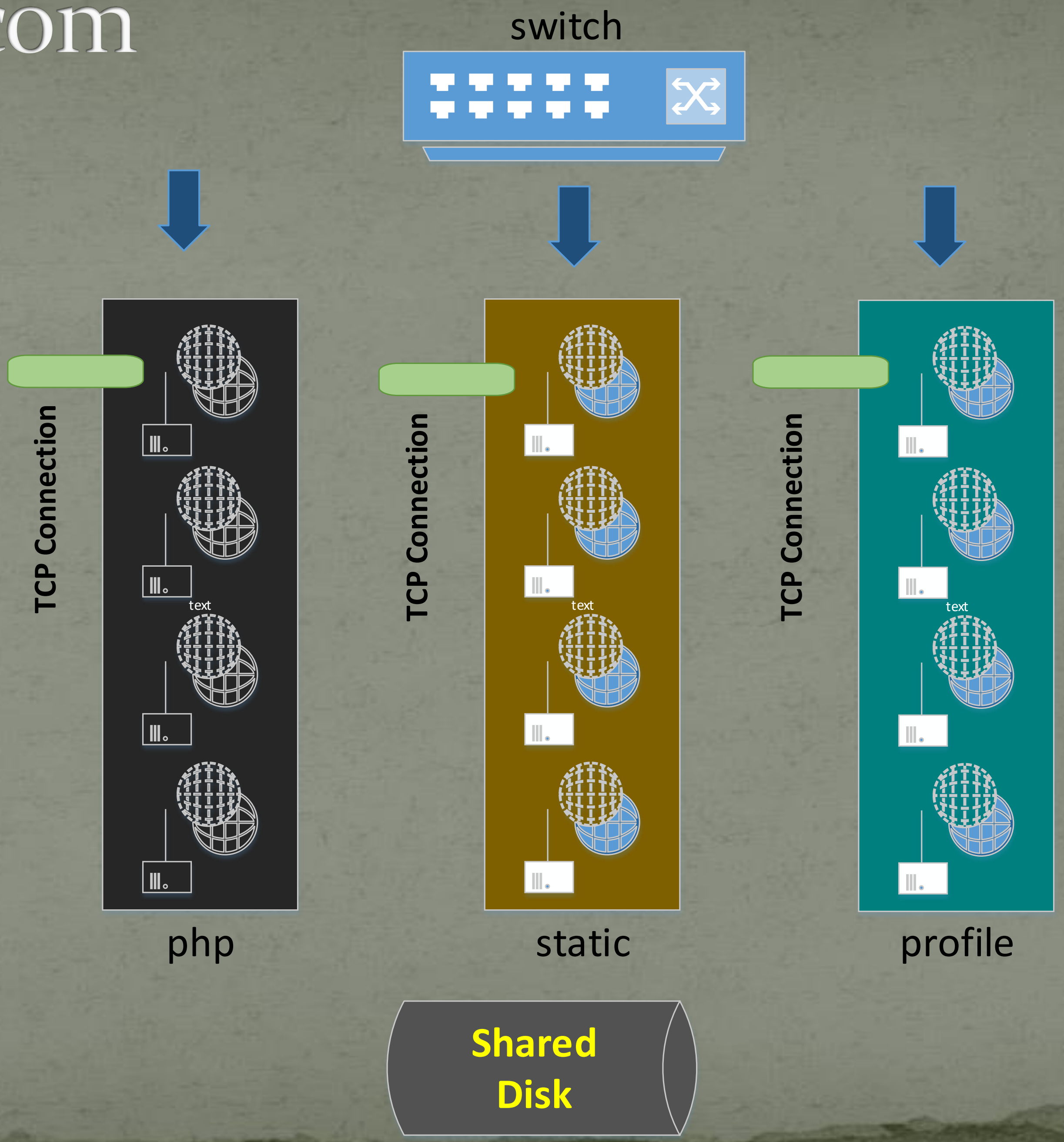
https:\\facebook.com

- **HTTP/1.1 parallelism requires using multiple, concurrent TCP sessions**



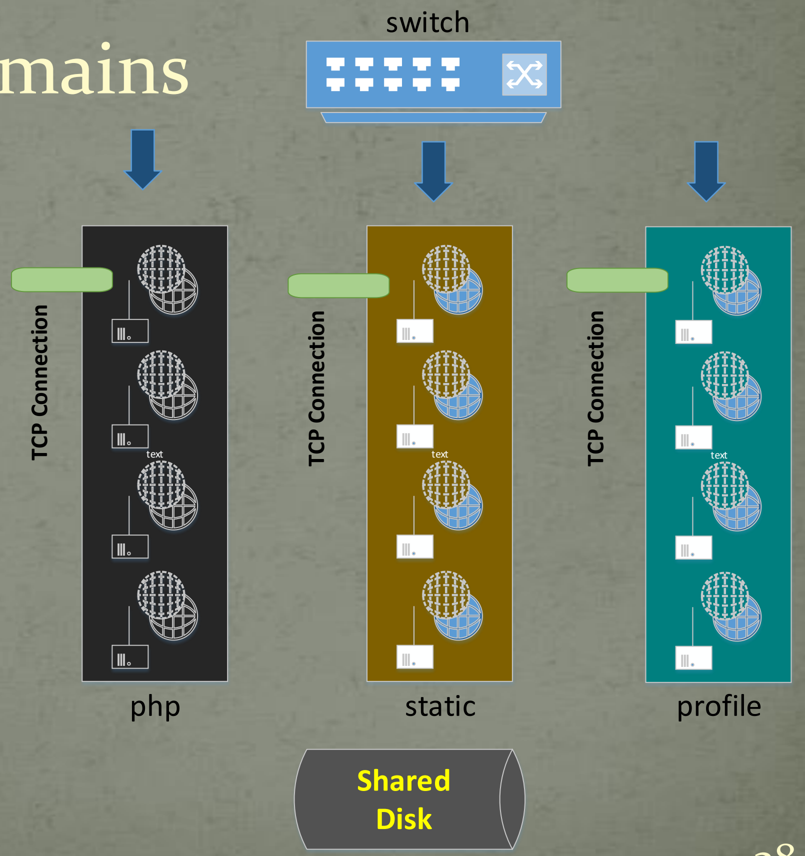
https:\\facebook.com

- current SPDY implementation:
- one TCP session per tier
- requires session-aware web servers at all three tiers
- not noticeably faster than HTTP/1



https:\\facebook.com

- **SPDY implementation not noticeably faster than HTTP/1.1 with parallel TCP sessions**
  - monolithic web site
  - (> 75% of the Requests ⇒ two Facebook domains
  - web servers must be session-aware
  - static content can be cached effectively
    - on the CDN
    - or in the web client



An Example: https:\\youtube.com

- Comparing HTTP/1.1 to SPDY/3 multiplexing

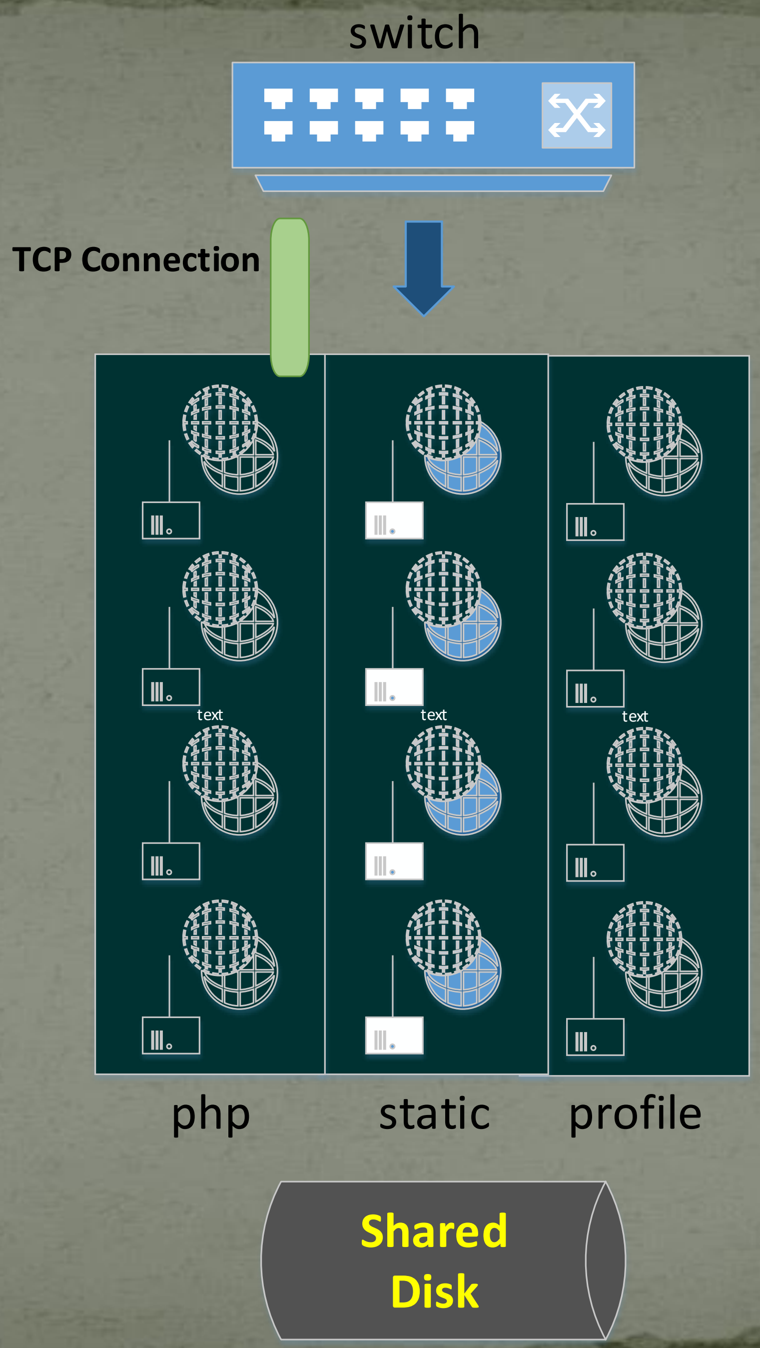
- 99 GET Requests ⇒ 4.4 MB landing page
  - Home page html: 500 KB
- Requests accounting for > 3 MB all directed to a single domain
  - 3 style sheets: 300 KB
  - JavaScript file for video playback: 900 KB
  - common.js library: 350 KB
  - 50 jpeg thumbnail images that serve as link buttons to the advertised videos
  - ten smaller graphic sprites, each 1.5-15 KB, from a second domain
  1. 5 JavaScript framework files from https://apis.google.com.
  - 10 JavaScript files from a 3rd domain
  - 10 small ads (~500 bytes each) from doubleclick (a Google web property)
  - 1 rich media display ad: 250 KB (from another Google web property)

Architecting for HTTP/2

- **Requires a new generation of web server software that knows how to consolidate Response messages into a single, session-oriented stream**
  - Responsive web design still required due to the wide variation in the capabilities of web clients/platforms
  - HTTP/2 changes do not impact native phone or tablet apps that call web services directly
- Consider TCP congestion control policy changes in order to maximize throughput over a single TCP connection

Architecting for HTTP/2

- **Eventually,**
  - Consolidating content on fewer domains should make web site administration easier
  - Undo any extreme web domain sharding that was done for HTTP/1.1
- **But that might be a whole bunch of web site re-engineering!**



Major new features in HTTP/2:

- Multiplexing
- Priority
- Server Push
- Header compression
- Improved performance with Transport Layer Security (compared to HTTPS)

- HTTP/2 requires changes at both the web client and web server

HTTP/2 Priority

- **Priority** would help web servers differentiate among multiple GET Requests sent by the web client
- Priority was not implemented in the SPDY experiment
- How HTML markup will indicate priority to the browser is currently undefined
  - e.g., Microsoft has been experimenting with a non-standard ***lazyload*** keyword in IE 10

HTTP/2 ***Server Push***

- ***Server Push*** would allow HHTP/2 web servers to send Response messages *before* specific GET Requests are received from the web client
- e.g.,
  - as soon as the initial Response message is handed to the TCP/IP stack for delivery
  - anticipating that the web client will making these Requests
  - the web server could start to push .css and image files referenced in the original Response message to the web client
- Goals:
  - improve line utilization
  - eliminate the need to ***inline*** scripts and style sheets for performance reasons

HTTP/2 *Server Push*

- ***Server Push*** specification
  - a new HTTP/2 frame called a *PUSH_PROMISE*
    - used by the web server to notify the client that it intends to push content into the interleaved Response message stream not yet Requested by the client.
  - Meanwhile, the web client might be searching its cache to locate the same HTTP object being pushed by the server
  - Web client can send a *RST_STREAM* message to reject the server push on a cache hit

  - Significant risk that PUSH_PROMISE and RST_STREAM messages could cross in the mail for cacheable, static content

HTTP/2 Header compression

- Primarily helps on uploads
  - The same Header data is sent for each GET Request in HTTP/1.1
    - cookie data
    - Host name and User Agent fields are sent in clear text
  - In HTTP/2,
    - the Server retains Header fields from earlier Requests
    - subsequent GET Requests to the same domain need only send added or changed Header fields
    - increases the number of GET Requests that require multiple packets
    - reduces the performance penalty associated with large cookies

HTTP/2 Security enhancements

- In HTTP/2, **improved performance with Transport Layer Security (TLS)**
  - unlike SPDY, does not require HTTPS
  - continues to plug into TCP Port 80
  - TLS can be requested at connection time
  - A fix that saves 2 handshaking packets to create a secure connection during the initial TCP session setup
- HTTP/2 also supports sending binary data fields in Request streams
  - binary data will initially present more challenges to hackers
  - But, expect they will quickly overcome this new obstacle

New feature summary for HTTP/2:

- **Multiplexing**
  - biggest change, but may require extensive web site re-engineering to take full advantage of
- **Server Push**
  - need to figure out the interaction with caching and CDNs
- **Priority**
  - need to understand the browser impact; will the DOM understand lazy loading of resources?
- **Header compression**
  - helps reduce the size of GET Request messages
  - requires additional web server changes to preserve header data between interactions
- **Improved performance with Transport Layer Security**
  - nice to have

- HTTP/2 bring significant changes to both the web client and web server, with the protocol embracing session-oriented behavior by default

HTTP/2 and TCP congestion control

- **HTTP/2 tries to push as many bytes as possible into the TCP Send Window of a connection as early and as often as possible.**
  - Maximize HTTP message throughput over a single TCP connection
- **Meanwhile, the TCP congestion control policy is conservative about overloading a connection**
  - ***slow start***, determines the small, initial size of the *cwin*
  - the size of the *cwin* ramps up slowly – ***additive increase***
  - backs off the transmission rate sharply when a congestion signal is received over a connection
    - ***multiplicative decrease***
    - the most common congestion signal is a *Send Window full* condition, corresponding to a Sender sending data faster than the Receiver can receive and process it

TCP congestion control

- **The conservative TCP congestion control policy**
  - initial size of the *cwin = 2 packets*
  - ***additive increase*** adds 1 packet to the *cwin* each Send interval
- So, for example,
  - over a connection with an **RTT = 100 ms**
  - maximum throughput = 10 * *cwin* / sec
  - during the first second of the connection:
    - *cwin* ranges from 2 – 11 * 1.5 KB pac kets
    - Sender can only transmit 55 packets, or about 80 KB

- In Windows, change the TCP defaults:

```
Set -NetTCPSetting -SettingName Custom
  -CongestionProvider CTCP
  -InitialCongestionWindowMss 16
```

TCP congestion control

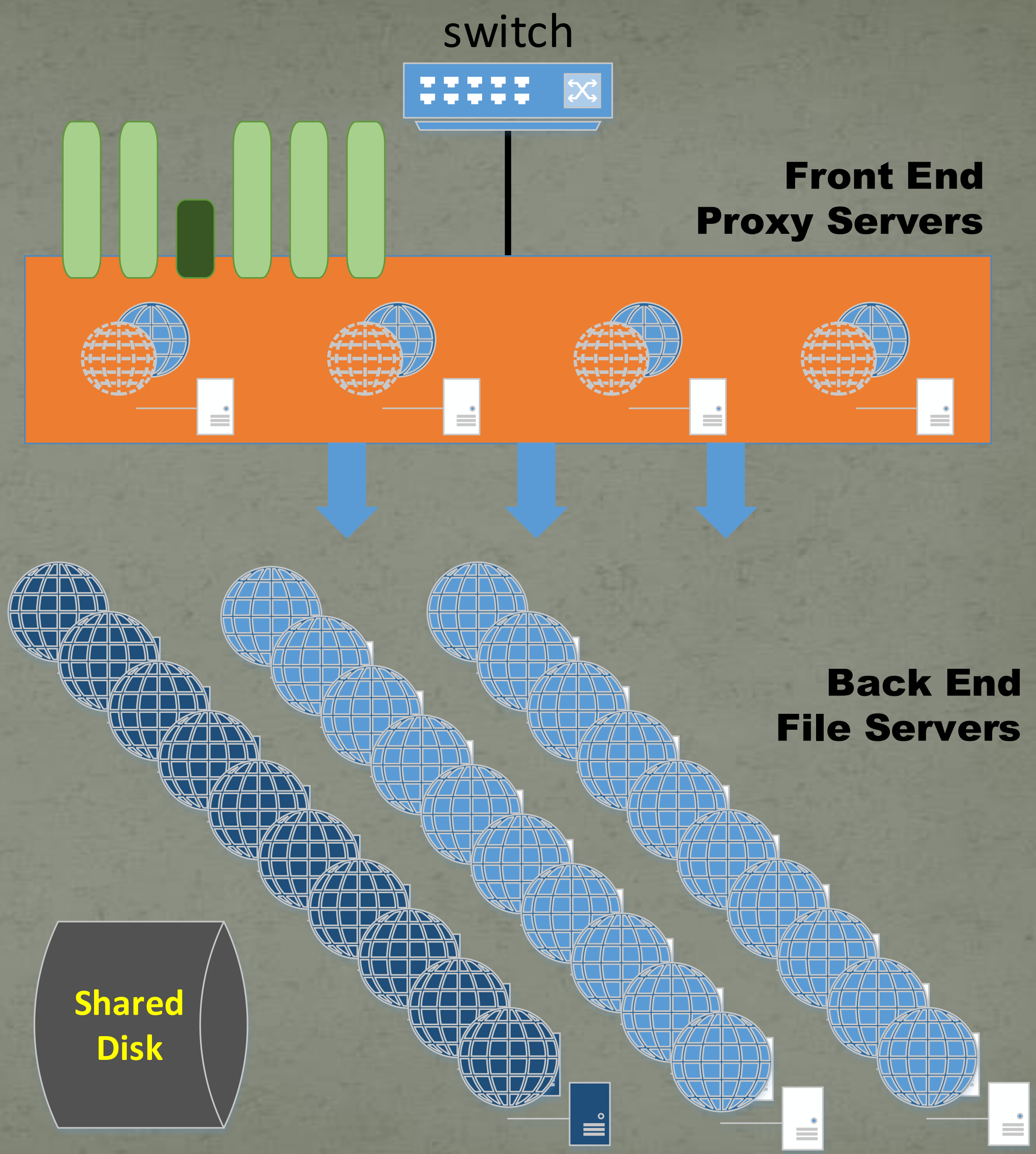
- **The conservative TCP congestion control policy**
  - on a congestion signal,
  - ***multiplicative decrease*** cuts the size of the *cwin* to *cwin* / 2
  - and reverts to *slow start*

- So, in HTTP/2 with one active TCP connection,
  - ***multiplicative decrease*** reduces the throughput over the connection by **50%**

- But, in HTTP/1.1 with parallel connections active between the client and server,
  - a single congestion signal has much less impact on overall throughput

TCP congestion control

- Impact of a congestion signal on a single connection is one of the reasons why SPDY does not consistently outperform a well-designed HTTP/1.1 web site



TCP congestion control

- **The conservative TCP congestion control policy**
  - ***multiplicative decrease*** sets the size of the cwin = *cwin* / 2 and reverts to *slow start*
  - Impact of a congestion single on a single connection is one of the reasons why SPDY does not consistently outperform a well-designed HTTP/1.1 web site

- In Windows, change the TCP defaults:

```
Set -NetTCPSetting –SettingName Custom
  –CwndRestart True
```

Summary

- HTTP/2 multiplexing is based on Google's SPDY experiment
- HTTP/2 makes the protocol more explicitly session-oriented, with implications for
  - the web server
  - the web client
  - web site re-engineering and re-architecture
- HTTP/2 throughput goals and default TCP congestion control policies are in conflict

Questions

?