HTTP in Depth

Vinayak Hegde
SANOG 30

Rough Outline

- **Understanding HTTP/1.1 Part 1 (90 minutes)**
  - Understanding Concepts from REST
  - Headers, Methods, Status codes (Demos+handson)
- **Understanding HTTP/1.1 Part 2 (90 minutes)**
  - Content negotiation (Demo)
  - Understanding Caching behavior
  - Real usage in the wild
    - Streaming use-cases (Demos)
    - HTTP APIs (Theory + Demos)
    - Discussion of implementations
    - Optimisations

Rough Outline

- **Understanding problems with HTTP/1.1 (45 minutes)**
  - Theory + Wireshark demos.
- **Motivations for HTTP/2 (45 minutes)**
  - Design Goals and Tradeoffs
  - Discussion on design choices
- **HTTP/2 in-depth (90 Minutes)**
  - HPACK Header compression
  - TLS + Security
  - Frames + Streams
  - Drawbacks on HTTP/2 and tradeoffs
  - Server Push

World before HTTP

- Precursors
  - Vannevar Bush “As we may think” - *Memex*
    - *“enlarged intimate supplement to one's memory”*
  - Ted Nelson – Project Xanadu
    - Contained some key ideas about hypertext but no implementation
- Gopher protocol
  - Archie / Veronica search engines
  - Based on file hierarchies

Lab 1

- Gopher Protocol fun
  - $ telnet gopher.floodgap.com
    - /
    - /gopher/welcome
  - Go to http://gopher.floodgap.com/

A Short history of HTTP

- Early 1990s - Tim Berners-Lee starts implementation of simple stateless protocol in CERN
- May 1996 - RFC 1945 – HTTP/1.0 specification
- Jan 1997 - RFC 2068 – HTTP/1.1 first specified
- Jun 1999 - RFC 2616 – HTTP/1.1 long lived draft (improved perfomance and tightens specification)
- The web explodes and the 2000s craze (and subsequent bust)
- June 2014 – Comprehensive overhaul based on real-world usage and issues - RFC 723[0-5]
- May 2015 – Starts with Google SPDY and ends with HTTP/2 – RFC 7540

Motivations for HTTP

- From RFC 7230
  - HTTP is a generic interface protocol for information systems. It is designed to hide the details of how a service is implemented by presenting a uniform interface to clients that is independent of the types of resources provided. Likewise, servers do not need to be aware of each client's purpose: an HTTP request can be considered in isolation rather than being associated with a specific type of client or a predetermined sequence of application steps. The result is a protocol that can be used effectively in many different contexts and for which implementations can evolve independently over time.

Motivations for HTTP

- From RFC 7230
  - HTTP is also designed for use as an intermediation protocol for translating communication to and from non-HTTP information systems. HTTP proxies and gateways can provide access to alternative information services by translating their diverse protocols into a hypertext format that can be viewed and manipulated by clients in thesame way as HTTP services.

HTTP/1.1 is a group of RFCs

- RFC 7230 - HTTP/1.1: Message Syntax and Routing
- RFC 7231 - HTTP/1.1: Semantics and Content
- RFC 7232 - HTTP/1.1: Conditional Requests
- RFC 7233 - HTTP/1.1: Range Requests
- RFC 7234 - HTTP/1.1: Caching
- RFC7235 - HTTP/1.1: Authentication

- These obsolete RFC 2616

Background IETF Info

- Worked on by HTTPBis working group
- Multiple contributors
- NOT Obsoleted by HTTP/2.0 (co-exists with it)
- Can be extended by other protocols like WEBDAV, HTTPS
- Other protocols can use Mappings (CoAP/QUIC?)
- Goto https://datatracker.ietf.org/wg/httpbis/documents/

List of Terms (RFC 7230)

- Client
- Server
- Proxy / Gateway (Reverse Proxy) / Tunnel
- User-agent
- MIME
- URI
- Cache

User agents - Lab 2

- Not all UA are web browsers
  - Household appliances
  - Mobile apps
  - Commandline programs
  - Firmware update scripts

```
$ curl -I http://httpbin.org/status/418
```

Resource

The target of an HTTP request is called a "resource". HTTP does not limit the nature of a resource; it merely defines an interface that might be used to interact with resources. Each resource is identified by a Uniform Resource Identifier (URI).

Detour on URI

- Defined by

  (eg. http://www.example.com:80/data?fmt=json)

  - Scheme (http)
  - host-uri (www.example.com)
  - Port (80)
  - Segment/path(/data)
  - Query (fmt=json)

  - RFC 3986 / RFC 7320 have more information

Lab 3 - URI

- Go to your favorite site and see the different parts of the URI and how they change behavior
- Application to HTTP API design

Messages

Each Hypertext Transfer Protocol (HTTP) message is either a request or a response. A server listens on a connection for a request, parses each message received, interprets the message semantics in relation to the identified request target, and responds to that request with one or more response messages. A client constructs request messages to communicate specific intentions, examines received responses to see if the intentions were carried out, and determines how to interpret the results.

Representation

For the purposes of HTTP, a "representation" is information that is intended to reflect a past, current, or desired state of a given resource, in a format that can be readily communicated via the protocol, and that consists of a set of representation metadata and a potentially unbounded stream of representation data.

Representation Metadata

- Content-Type
- Content-Encoding
- Content-Language
- Content-Location

Content Type

- The "Content-Type" header field indicates the media type of the associated representation: either the representation enclosed in the message payload or the selected representation, as determined by the message semantics.
- Default may be "application/octet-stream"
- Security risks due to content sniffing and misinterpretation
- eg. text/html;charset=utf-8
- Uses MIME Types but does not support all

Content Encoding

- Content-Encoding: eg. gzip - used for compression
- Different from Transfer-encoding – is a property of the resource and not the transport
- 415 (Unsupported Media Type)

Lab 4 – Demo of httpie

- On ubuntu
  - $ sudo apt-get install httpie
  - $ http zappos.com
  - $ http www.zappos.com
  - Suggest other sites

Content Language

- Content languages syntax and registry listed in RFC 5646
- Content Language identifies a natural language spoken, written, or otherwise conveyed by human beings for communication of information to other human beings.
- Computer languages are explicitly excluded.
- Can contain multiple languages and language specification cannot contain spaces.

Content Location

- Content-Location - Contains URIs and the current representation in the messages
- Avoids a GET request to know content has changes
- 201 response shows that resource has been created
- Could also tell for a 200 response that receipt is at a different location

Request Semantics

The request method token is the primary source of request semantics; it indicates the purpose for which the client has made this request and what is expected by the client as a successful result.

Methods (Most common)

- GET - Transfer a current representation of the target resource
- HEAD - Same as GET, but only transfer the status line and header section
- POST - Perform resource-specific processing on the request payload
- PUT - Replace all current representations of the target resource with the request payload.

Methods (less common)

- DELETE - Remove all current representations of the target resource.
- CONNECT - Establish a tunnel to the server identified by the target resource.
- OPTIONS - Describe the communication options for the target resource.
- TRACE - Perform a message loop-back test along the path to the target resource.

Methods

HTTP Method	RFC	Request Has Body	Response Has Body	Safe	Idempotent	Cacheable
GET	RFC 7231	No	Yes	Yes	Yes	Yes
HEAD	RFC 7231	No	No	Yes	Yes	Yes
POST	RFC 7231	Yes	Yes	No	No	Yes
PUT	RFC 7231	Yes	Yes	No	Yes	No
DELETE	RFC 7231	No	Yes	No	Yes	No
CONNECT	RFC 7231	Yes	Yes	No	No	No
OPTIONS	RFC 7231	Optional	Yes	Yes	Yes	No
TRACE	RFC 7231	No	Yes	Yes	Yes	No
PATCH	RFC 5789	Yes	Yes	No	No	Yes

Notes about Method

- All general-purpose servers MUST support the methods GET and HEAD.
- All other methods are OPTIONAL.
- Additional methods, outside the scope of this specification, have been standardized for use in HTTP. All such methods ought to be registered within the "Hypertext Transfer Protocol (HTTP) Method
- Notes
  - 501 (Not Implemented)
  - 405 (Method Not Allowed)

Notes About the PUT method

- HTTP does not define exactly how a PUT method affects the state of an origin server beyond what can be expressed by the intent of the user agent request and the semantics of the origin server response.
- Resource Content type configured different than what PUT mentions then:
  - reconfigure the target resource to reflect the new media type;
  - transform PUT representation to a format consistent with that of resource before saving it as the new resource state;
  - reject the request with a 415 (Unsupported Media Type) response indicating that the target resource is limited to "text/html", perhaps including a link to a different resource that would be a suitable target for the new representation.

Difference between PUT and POST

The fundamental difference between the POST and PUT methods is highlighted by the different intent for the enclosed representation.The target resource in a POST request is intended to handle the enclosed representation according to the resource's own semantics, whereas the enclosed representation in a PUT request is defined as replacing the state of the target resource. Hence, the intent of PUT is idempotent and visible to intermediaries, even though the exact effect is only known by the origin server.

Note about CONNECT

- The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request target and, if successful, thereafter restrict its behavior to blind forwarding of packets, in both directions, until the tunnel is closed. Tunnels are commonly used to create an end-to-end virtual connection, through one or more proxies, which can then be secured using TLS (Transport Layer Security, [RFC5246]).
- Security considerations
  - Authentication
  - ACL
  - Whitelist

Break

Lab : Tools

- A short intro to tools
  - Curl
  - Wireshark
  - Livehttpheaders / ChromeDevTools

Lab

- Play with httpbin.org
- https://requestb.in/

Wireshark

Content Negotiation

- This RFC defines two patterns of content negotiation that can be made visible within the protocol:
  - **"proactive"**, where the server selects the representation based upon the user agent's stated preferences, and
  - **"reactive"** negotiation, where the server provides a list of representations for the user agent to choose from.

Proative Negotiation

- Parameters for selection
  - Explicit – Accept, Accept-Charset, Accept-Encoding, Accept-Language
  - Implicit - Network Address or UA
- Downsides
  - What is "best" for the user
  - Efficiency of requests
  - Complicates server implementation
  - Affects caching behavior
- Note: A Vary header field (Section 7.1.4) is often sent in a response subject to proactive negotiation to indicate what parts of the request information were used in the selection algorithm.

Reactive Negotiation

- Downsides
  - User latency
  - Not speced out so server/client combination dependent so could have interoperability issues

Note about Content/Resource

HTTP is not aware of the resource semantics. The consistency with which an origin server responds to requests, over time and over the varying dimensions of content negotiation, and thus the "sameness" of a resource's observed representations over time, is determined entirely by whatever entity or algorithm selects or generates those responses. HTTP pays no attention to the man behind the curtain.

Content Negotiation

- Accept
- Accept-Charset
- Accept-Encoding
- Accept-Language
- Examples
  - Accept: text/plain; q=0.5, text/html,text/x-dvi; q=0.8,text/x-c
  - Accept-Charset: iso-8859-5, unicode-1-1;q=0.8
  - Accept-Language: da, en-gb;q=0.8, en;q=0.7

Content Negotiation demo

Response Codes

- The status-code element is a three-digit integer code giving the result of the attempt to understand and satisfy the request.
- HTTP status codes are extensible. HTTP clients are not required to understand the meaning of all registered status codes, though such understanding is obviously desirable. However, a client MUST understand the class of any status code, as indicated by the first digit, and treat an unrecognized status code as being equivalent to the x00 status code of that class, with the exception that a recipient MUST NOT cache a response with an unrecognized status code.

Categories of Response Code

- The first digit of the status-code defines the class of response. The last two digits do not have any categorization role. There are five values for the first digit:
  - 1xx (Informational): The request was received, continuing process
  - 2xx (Successful): The request was successfully received, understood, and accepted
  - 3xx (Redirection): Further action needs to be taken in order to complete the request
  - 4xx (Client Error): The request contains bad syntax or cannot be fulfilled
  - 5xx (Server Error): The server failed to fulfill an apparently valid request

Response codes cheat sheet

- 1xx – Hold on
- 2xx – Here you go
- 3xx – Go Away
- 4xx – You messed up
- 5xx – Looks like I messed up

Selected Response codes (Demo)

- 100 Continue
- 201 Created
- 202 Accepted
- 400 Bad Request
- 403 Forbidden
- 404 Not Found
- 409 Conflict
- 414 URI Too Long
- 500 Internal Server Error
- 504 Gateway Timeout

Note About 307 (Temporary redirect)

Note: This status code is similar to 302 (Found), except that it does not allow changing the request method from POST to GET. This specification defines no equivalent counterpart for 301 (Moved Permanently) ([RFC7238], however, defines the status code 308 (Permanent Redirect) for this purpose).

Considerations for new status codes

When it is necessary to express semantics for a response that are not defined by current status codes, a new status code can be registered. Status codes are generic; they are potentially applicable to any resource, not just one particular media type, kind of resource,Or application of HTTP. As such, it is preferred that new status codes be registered in a document that isn't specific to a single application.

Considerations for New Header Fields

- Authors of specifications defining new fields are advised to keep the name as short as practical and not to prefix the name with "X-" unless the header field will never be used on the Internet. (The "X-" prefix idiom has been extensively misused in practice; it was intended to only be used as a mechanism for avoiding name collisions inside proprietary software or intranet processing, since the prefix would ensure that private names never collide with a newly registered Internet name; see [BCP178] for further information).

A note about Range Requests

- Defined in RFC 7233
- Optional to implement in clients
- Used for streaming cases
- Used for download resumes
- Behaviour of servers in range requests

Range Requests

- Accept-Ranges Header
- Content-Ranges Header
- If-Range Header
- Range Header
- Response Code: 206 Partial Content
- Response Code: 416 Range Not Satisfiable
- Denial-of-Service Attacks Using Range

A Note on Caching

- Caching is defined in RFC 7234
- Optional feature
- Intermediate caches
- Browser caches
- Should not store auth request/responses

Cache Requests

- “No-store” directive
- “Private” directive
- Expires Header
- Max-age header
- S-maxage header

Cache request

- Cache freshness algorithm
- Interaction with timezones
- If-modified-since Header
- If-unmodfied-since Header
- Response Code: 304 (Not modifed)
- Freshening Responses via HEAD
- Invalidation and interaction with PUT & POST
- Usage of Etags

Cache Control Headers

- Max-age
- Max-stale
- Min-Fresh
- No-Cache
- No-Store
- No-transform
- Must-revalidate
- Warning

A Note about Reverse Proxies

- A reverse proxy is mostly a server-side concept, and is usually used in the context of CDNs (content distribution networks) for caching static HTTP content.
- A forward proxy is usually a client side concept used for anonymity, to subvert censorship, and (back in the days of dial-up) as a web accelerator.
- eg. Akamai, CloudFlare, Varnish

Uses of Reverse Proxy Server

- **Load balancing** – A reverse proxy server can act as a “traffic cop,” sitting in front of your backend servers and distributing client requests across a group of servers in a manner that maximizes speed and capacity utilization while ensuring no one server is overloaded, which can degrade performance. If a server goes down, the load balancer redirects traffic to the remaining online servers.
- **Web acceleration** – Reverse proxies can compress inbound and outbound data, as well as cache commonly requested content, both of which speed up the flow of traffic between clients and servers. They can also perform additional tasks such as SSL encryption to take load off of your web servers, thereby boosting their performance.
- **Security and anonymity** – By intercepting requests headed for your backend servers, a reverse proxy server protects their identities and acts as an additional defense against security attacks. It also ensures that multiple servers can be accessed from a single record locator or URL regardless of the structure of your local area network.

Break

Lab

- Experimental features and network internals
  - Firefox about:config
  - Chrome about:about

Security considerations for HTTP/1.1

- Attacks Based on File and Path Names
  - Relative path
  - Wildcards
  - Special files
- Attacks Based on Command, Code, or Query Injection
  - Validation of user data
  - Dont trust user data / escape it
  - Whitelist plugins if necessary
- Disclosure of Personal Information
- Disclosure of Sensitive Information in URIs
- Disclosure of Fragment after Redirects
- Disclosure of Product Information
- Browser Fingerprinting

HTTP Optimisations Demo

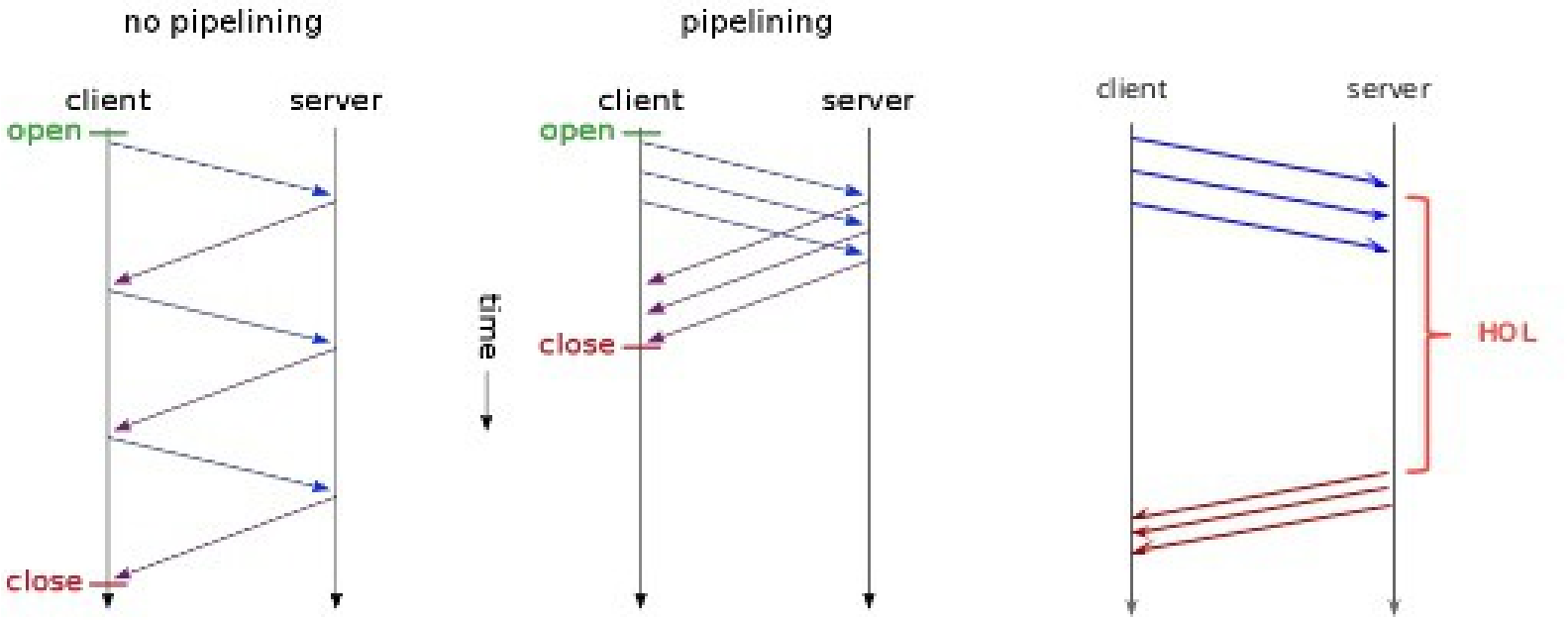
- Chrome devtools
- Webpagetest (https://www.webpagetest.org/)

HTTP Optimisations

- HTTP pipelining / Head-of-line blocking
- Domain Sharding
- Spriting of images
- Cookie-free domains
- File concatenation
- Resource Inlining
- Compression (gzip, deflate, brotli)
- Browser painting (Not all resources are equal)

HTTP pipelining detour

HTTP does not support multiplexing!



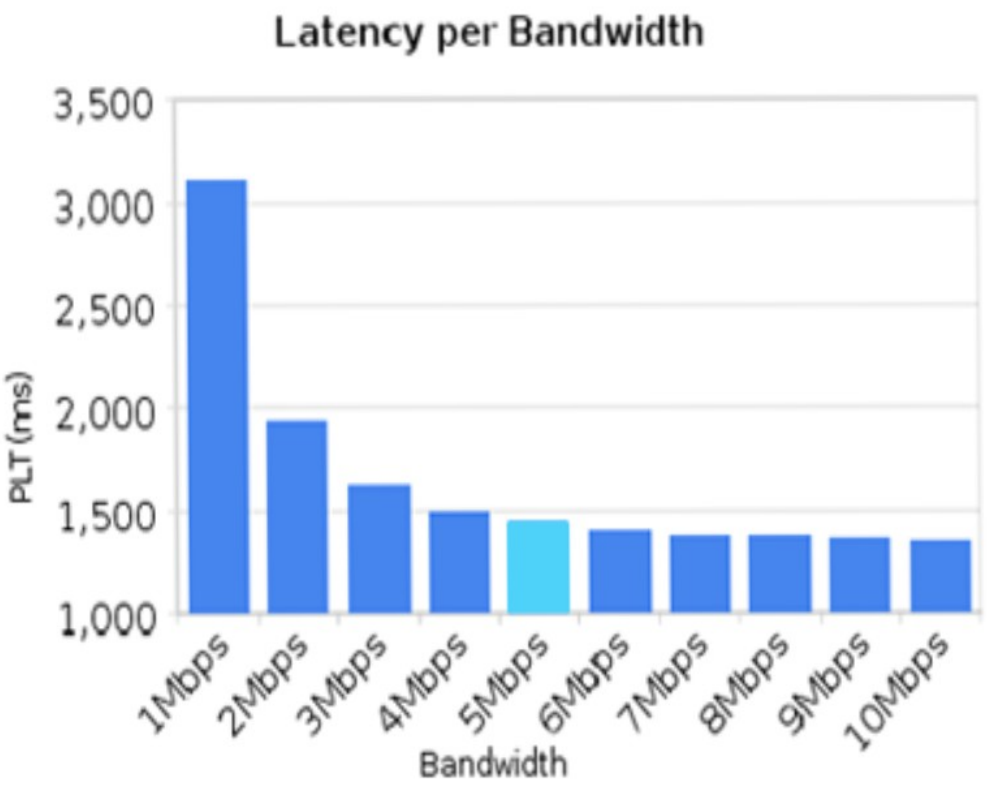
- **No pipelining:** request queuing
- **Pipelining*:** response queuing

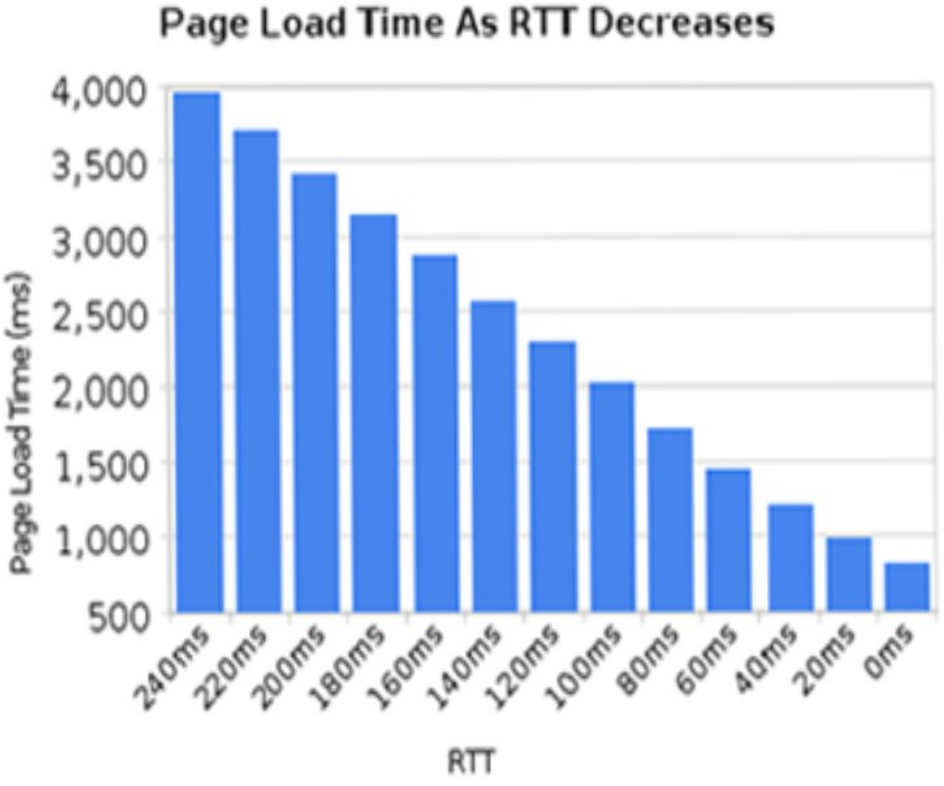
- **Head of Line blocking**
  - It's a guessing game...
  - Should I wait, or should I pipeline?

@igrigorik

Ref : https://www.slideshare.net/heavybit/heavybit-presents-ilya-grigorik-on

Bandwidth Vs Latency





Ref :https://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck/

Problems with HTTP/1.1

- Web developers have worked around HTTP's limitations, but:
  - Performance still falls short of full bandwidth utilization
  - Web design and maintenance are more complex
  - Resource consumption increases for client and server
  - Cacheability of resources suffers

Lab HTTParchive

- http://httparchive.org/

So how do we make HTTP better ?

- What if we could redesign http for the modern age (fast networks, low latency)
  - It would be less sensitive to network delay
  - fixed pipelining and HOL blocking
  - performed well regardless of number of tcp connections used
  - had long lived connections to take better advantage TCP's congestion control to kick in
  - used the same semantics as http/1.1

A Brief History of HTTP/2

- November 2009: Mike Belshe and Roberto Peon announce SPDY
- March 2011: Mike talks about SPDY to the HTTPbis WG at IETF80
- ~April 2011: Chrome, Google start using SPDY
- March 2012: HTTPbis solicits proposals for new protocol work
- March 2012: Firefox 11 ships with SPDY (off by default)
- May 2012: Netcraft finds 339 servers that support SPDY
- June 2012: Nginx announces SPDY implementation
- July 2012: Akamai announces SPDY implementation
- July 2012: HTTPbis re-chartered to work on HTTP/2.0, based on SPDY

HTTP2 Support

- Now supported by all major browsers – Firefox, Chrome, Safari, Opera and MS Edge
- Also supported by major HTTP servers - Nginx, Apache, Lightspeed, Microsoft IIS and HAProxy
- Supported by Major CDNs – Akamai, Cloudflare, Fastly and AWS Cloudfront

Important Points

- NO change to HTTP semantics; it's about how it gets onto the wire
- You will not need to change your websites or applications to ensure they continue to work properly. Not only will your application code and HTTP APIs continue to work uninterrupted, but your application will also likely perform better and consume fewer resources on both client and server.
- HTTP/1.1 and HTTP/2 are likely to coexist for a long time
- HTTP/2 builds on top of HTTP/1.1
- HTTP/2 or h2 does not require encryption in practice but in reality almost all clients implement it (including all major browsers).

HTTP/2 features

- **Multiplexing and concurrency**: Several requests can be sent in rapid succession on the same TCP connection, and responses can be received out of order - eliminating the need for multiple connections between the client and the server
- **Stream dependencies**: the client can indicate to the server which of the resources are more important than the others
- **Header compression**: HTTP header size is drastically reduced
- **Server push**: The server can send resources the client has not yet requested
- **“Upgrade:” header** or TLS ALPN negotiation

Considerations for HTTP/2

- **Encryption**: Applications running over HTTP/2 are likely to experience improvements in performance over secure connections. This is an important consideration for companies contemplating the move to TLS.
- **Optimizing the TCP layer**: Applications should be designed with a TCP layer implemented to account for the switch from multiple TCP connections to a single long-lived one, especially when adjusting the congestion window in response to packet loss.

Considerations for HTTP/2

- **Undoing HTTP/1.1 best practices**: Many "best practices" associated with applications delivered over HTTP/1.1 (such as domain sharding, image spriting, resource in-lining and concatenation) are not only unnecessary when delivering over HTTP/2, and in some cases may actually cause sub-optimizations.
- **Deciding what and when to push**: Applications designed to take advantage of the new server push capabilities in HTTP/2 must be carefully designed to balance performance and utility.

HTTP Optimisations Demo

- Chrome devtools
- Webpagetest (https://www.webpagetest.org/)

Lab : Is H2 really faster ?

- Lets try it
  - https://http2.akamai.com/demo
  - https://http2.golang.org/

HTTP2 Concepts

- HTTP2 is a binary protocol
- Concept of frames
- Frames sent from server to browser and vice-versa
- Concept of Streams
- Streams are multiplexed with priorities

More about HTTP2

- Defined in RFC 7540
- Longer lived connections and hence better congestion control
- Fairness is an consideration for shared network

Types of Streams

- **DATA** - Convey Arbitrary data associated with stream
- **HEADERS** - Used to open a stream and carries name:value pairs
- **PRIORITY** - Specifies sender-advised priority of streams
- **RST_STREAM** - Allows abnormal termination of stream
- **SETTINGS** - Conveys configuration parameters that affect how end-points communicate

Types of Streams

- **PUSH-PROMISE** - Used to notify the peer endpoint in advance of streams the sender intends to initiate
- **PING** - Measuring a minimal round-trip time from the sender; checks if a connection is still alive
- **GOAWAY** - Informs the remote peer to stop creating streams on this connection
- **WINDOW_UPDATE** - Used to implement flow control on each individual stream or on the entire connection.
- **CONTINUATION** - Used to continue a sequence of header block fragments

A note about Header Compression

- Defined in RFC 7451
- HPACK - Header compression algorithm
- HPACK has been invented because of attacks like CRIME and BREACH attacks
-

HPACK Stats

- Cloudflare saw an 76% compression for ingress headers and 53% drop in ingress traffic due to HPACK (Request traffic)
- Cloudflare saw a 69% compression for egress headers and 1.4% drop in egress traffic (Response traffic)

(Ref https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/)

Lab : nghttp2

- $ nghttp -nv https://nghttp2.org
- $ nghttp -nv http://http2.golang.org
- $ nghttp -nv http://vinayakhegde.com:5000
- $ nghttp -nv http://http2.akamai.com:5000

Lab Go http2 server

How HPACK Works - Dictionaries

- 3 types of compression
  - ***Static Dictionary***: A predefined dictionary of 61 commonly used header fields, some with predefined values.
  - ***Dynamic Dictionary***: A list of actual headers that were encountered during the connection. This dictionary has limited size, and when new entries are added, old entries might be evicted.
  - ***Huffman Encoding***: A static Huffman code can be used to encode any string: name or value. This code was computed specifically for HTTP Response/Request headers - ASCII digits and lowercase letters are given shorter encodings. The shortest encoding possible is 5 bits long, therefore the highest compression ratio achievable is 8:5 (or 37.5% smaller)

How HPACK works - Algorithm

- When HPACK needs to encode a header in the format name:value, it will first look in the static and dynamic dictionaries.
- If the full name:value is present, it will simply reference the entry in the dictionary. This will usually take one byte, and in most cases two bytes will suffice. A whole header encoded in a single byte.
- Since many headers are repetitive, this strategy has a very high success rate. (like long cookie headers)

How HPACK works - Algorithm

- When HPACK can't match a whole header in a dictionary, it will attempt to find a header with the same name.
  - Most of the popular header names are present in the static table, for example: content-encoding, cookie, etag.
  - The rest are likely to be repetitive and therefore present in the dynamic table.
  - Cloudflare assigns a unique cf-ray header to each response, and while the value of this field is always different, the name can be reused.

A note about ALPN

- A TLS extension that permits the application layer to negotiate protocol selection within the TLS handshake.
- Defined in RFC 7301
- With ALPN, the client sends the list of supported application protocols as part of the TLS ClientHello message. The server chooses a protocol and sends the selected protocol as part of the TLS ServerHello message. The application protocol negotiation can thus be accomplished within the TLS handshake, without adding network round-trips, and allows the server to associate a different certificate with each application protocol, if desired.

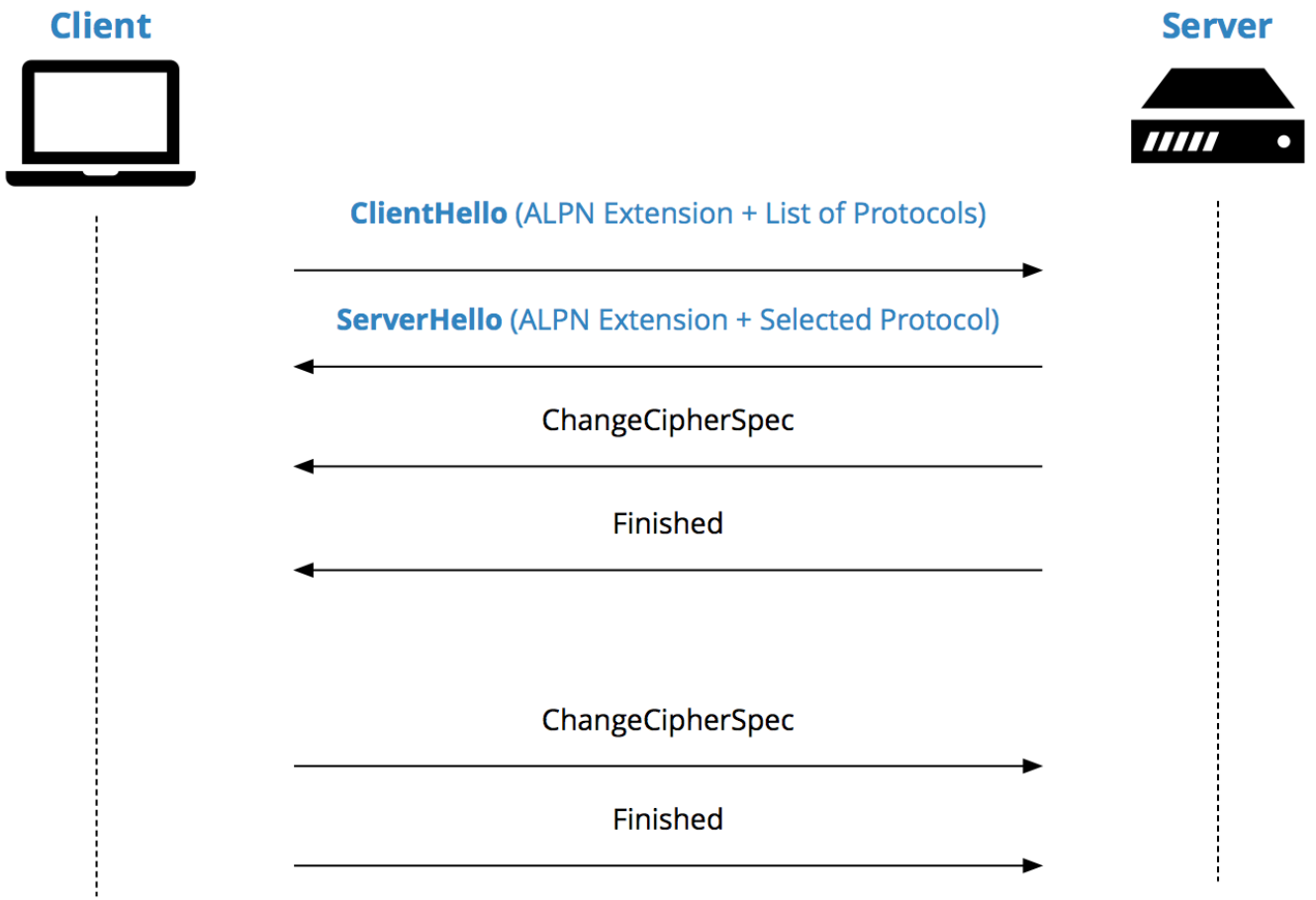
Why ALPN ?

- When multiple application protocols are supported on a single server-side port number, such as port 443, the client and the server need to negotiate an application protocol for use with each connection. It is desirable to accomplish this negotiation without adding network round-trips between the client and the server, as each round-trip will degrade an end-user's experience. Further, it would be advantageous to allow certificate selection based on the negotiated application protocol. (RFC 7301)

How TLS works (RFC 5246)

- The Client and Server exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.
- An exchange of the necessary cryptographic parameters allow the client and server to agree on a premaster secret.
- A master secret is generated from the premaster secret and exchanged random values.
- Security parameters are provided to the record layer.
- The Client and server verify that their peer has calculated the same security parameters and that the handshake took place without tampering by an attacker.

How ALPN works (RFC 7301)



More notes about ALPN

- The ALPN extension does not impact the security of TLS session establishment or application data exchange.

Demo of HTTP2

- Peaking at the internals of HTTP2 using Chrome and WireShark

Lab Shodan demo : H2 adoption

How many services support HTTP/2?

$ shodan count ssl.alpn:h2

How about SPDY?

$ shodan count ssl.alpn:spdy

Show a breakdown of the most popular HTTP versions

$ shodan stats --facets ssl.alpn has_ssl:true

Show the top 100 versions

$ shodan stats --facets ssl.alpn:100 has_ssl:true

Lets take a look at the organizations that support HTTP/2

achillean@demo:~$ shodan stats --facets org ssl.alpn:h2

Security considerations for HTTP2

- Server Authority
- Cross-Protocol Attacks
- Intermediary Encapsulation Attacks
- Cacheability of Pushed Responses
- Denial-of-Service Considerations
- Use of Compression
- Use of Padding
- Privacy Considerations

Tradeoffs and drawbacks

- Security is hard to get right
- TCP is awkward
  - In-order delivery = head-of-line blocking
  - Initial congestion window is small
  - Packet loss isn’t handled well
- Binary protocol so debugging is hard(er)

Thank you

- Resources