Middleware Architectures 1
Lecture 4: HTTP/2

doc. Ing. Tomáš Vitvar, Ph.D.

tomas@vitvar.com • @TomasVitvar • https://vitvar.com

Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • https://vitvar.com/lectures

Overview

- Introduction
- HTTP/2
- HTTP/3

Overview

- Developed from SPDY (2009) experimental protocol by Google
  - *May 2015: RFC 7540 (HTTP/2) and RFC 7541 (HPACK)*
  - *HTTP/2 standards extend (not replace) the previous HTTP standards*
- HTTP/1.x limitations
  - *HTTP/1.x clients need to use multiple connections to achieve concurrency*
  - *unnecessary network traffic – request and response headers not compressed*
  - *no effective resource prioritization*
- Primary goals
  - *Reduction of latency*
  - *enabling full request and response multiplexing*
  - *minimize protocol overhead via efficient compression of HTTP header fields*
  - *support for request prioritization and server push*
- HTTP/2 does not modify application semantics of HTTP
  - *HTTP metods, URIs, header fields are the same*
  - *HTTP/2 modifies **how data is formatted and transported** in communication*
- Literature and source
  - *I. Grigorik: High Performance Browser Networking, O'Reilly Media, Inc. 2013. ISBN: 9781449344757*

Establishing a HTTP/2 Connection

- Negotiating HTTP/2 via a secure connection with TLS and ALPN
  - *Client sends a protocol (HTTP/2) in a TLS* `ClientHello` *message.*

- Upgrading a plaintext connection to HTTP/2 without prior knowledge
  - *Client starts HTTP/1.1 and then sends an upgrade request*

```
 1   GET /page HTTP/1.1
 2   Host: server.example.com
 3   Connection: Upgrade, HTTP2-Settings
 4   Upgrade: h2c
 5   HTTP2-Settings: (SETTINGS payload)
 6
 7   HTTP/1.1 200 OK
 8   Content-length: 243
 9   Content-type: text/html
10
11   (... HTTP/1.1 response ...)
12
13           (or)
14
15   HTTP/1.1 101 Switching Protocols
16   Connection: Upgrade
17   Upgrade: h2c
18
19   (... HTTP/2 response ...)
```

- Initiating a plaintext HTTP/2 connection with prior knowledge
  - *Client gets information about server's HTTP/2 via DNS or manual configuration.*
  - *Client initiates HTTP/2 and if it does not work, it falls back to HTTP/1.1*

Overview

- Introduction
- HTTP/2
  - *Binary Framing*
  - *Stream Prioritization*
  - *Flow Control*
  - *Server Push*
  - *Header Compression*
  - *HTTP/2 Analysis*
- HTTP/3

Binary Framing Layer

- Binary framing layer
  - *defines how HTTP messages are encapsulated and transferred*
 - *communication is split into messages and frames in binary format*

HTTP/2 Communication

- Data exchange between the client and server
  - *break down of the communication into frames*
  - *frames are mapped to messages that belong to a particular stream*
  - *communication is **multiplexed within a single TCP connection**.*

- Stream
  - *bi-directional flow of bytes in a connection*
  - *may carry one or more messages*
  - *may have a priority*

- Message
  - *a sequence of frames*
  - *it maps to logical request or response message*

- Frame
  - *the smallest unit of communication*
  - *each has a frame header which identifies a stream to which it belongs.*

Streams, Messages, and Frames
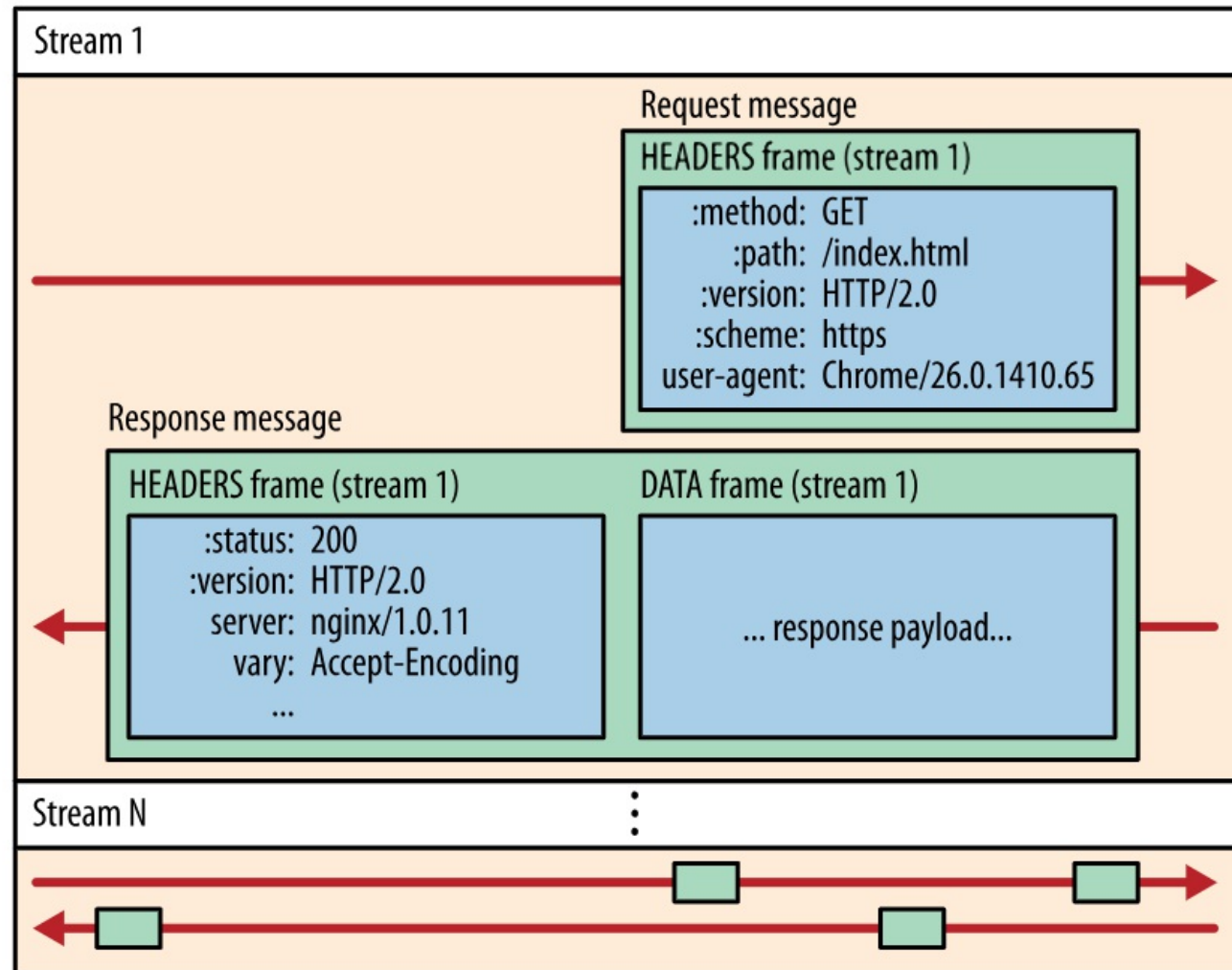
Request and Response Multiplexing

- Parallel requests
  - *HTTP/1.x can use HTTP pipelining; they open multiple connections*
    - → *browser typically opens up six connections*
    - → *One response can be delivered at a time (response queuing) per connection*
      - → *Head of line blocking problem*
  - *HTTP/2 allows full request and response multiplexing*
    - → *Allows for parallel in-flight streams*
    - → *There are 3 parallel streams in the below example:*

HTTP 2.0 connection

stream 1 DATA	stream 3 HEADERS	stream 3 DATA	stream 1 DATA	...

stream 5 DATA

Client   Server

- Performance benefits
  - *Interleave requests/responses in parallel without blocking any one.*
  - *Deliver lower page load times by eliminating unnecessary latency*
  - *Improve utilization of available network capacity*

Binary Framing – Frame Header

- 9-byte frame header

Bit	+0..7	+8..15	+16..23	+24..31
0	Length			Type
32	Flags			
40	R	Stream Identifier		
...	Frame Payload			

- Fields
  - `length` – *24 bits, allows a frame to carry 2^{24} bytes of data.*
  - `type` – *8 bits, determines the format and semantics of the frame.*
    - → *Frame types:* DATA, HEADERS, PRIORITY, RST_STREAM, SETTINGS, PUSH_PROMISE, PING, GOAWAY, WINDOW_UPDATE, CONTINUATION
  - `flags` – *8 bits, defines frame-type specific boolean flags.*
  - `stream identifier` – *31 bits, uniquely identifies the HTTP/2 stream.*

Stream Lifecycle

- HTTP/2 doesn't reuse the same stream IDs
  - *A stream's lifecycle corresponds to request/response messages interaction.*
- A new id is assigned until it reaches 2^{31}
  - *When the last id is used, the browser sends* `GOAWAY` *frame to initialize a new TCP connection, and the stream ID is reset.*

Initiating a New Stream

```
▼ HyperText Transfer Protocol 2
   ▼ Stream: HEADERS, Stream ID: 1, Length 20
      Length: 20
      Type: HEADERS (1)
   ▼ Flags: 0x05
         .... ...1 = End Stream: True
         .... .1.. = End Headers: True
         .... 0... = Padded: False
         ..0. .... = Priority: False
         00.0 ..0. = Unused: 0x00
      0... .... .... .... .... .... .... .... = Reserved: 0x00000000
      .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
      [Pad Length: 0]
      Header Block Fragment: 8682418aa0e41d139d09b8f01e078453032a2f2a
      [Header Length: 100]
   ▶ Header: :scheme: http
   ▶ Header: :method: GET
   ▶ Header: :authority: localhost:8080
   ▶ Header: :path: /
   ▼ Header: accept: */*
         Name Length: 6
         Name: accept
         Value Length: 3
         Value: */*
         Representation: Literal Header Field with Incremental Indexing — Indexed Name
         Index: 19
```

common frame header

HPACK encoded headers

– *New stream created with request metadata*
– *HEADERS and DATA frames sent separately*

Sending Application Data

```
▼ HyperText Transfer Protocol 2
   ▼ Stream: DATA, Stream ID: 1, Length 5
       Length: 5
       Type: DATA (0)
     ▼ Flags: 0x00
         .... ...0 = End Stream: False
         .... 0... = Padded: False
         0000 .00. = Unused: 0x00
       0... .... .... .... .... .... .... .... = Reserved: 0x00000000
       .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
       [Pad Length: 0]
       Data: 48656c6c6f
```

```
0000   02 00 00 00 45 00 00 42   89 06 40 00 40 06 00 00   ....E..B ..@.@...
0010   7f 00 00 01 7f 00 00 01   1f 90 d8 eb 8a 94 78 19   ........ ......x.
0020   7d b6 67 50 80 18 23 dd   fe 36 00 00 01 01 08 0a   }.gP..#. .6......
0030   6a 78 1f ec 6a 78 1f ec   00 00 05 00 00 00 00 00   jx..jx.. ........
0040   01 48 65 6c 6c 6f                                   .Hello
```

   — *App data sent in DATA frame*

   — *There are more frames that carry the data (i.e.* END_STREAM *flag is not set)*

      → *The small frame size allows for efficienct multiplexing*

   — *The app data is loaded by application according to the used encoding mechanism (plain text, gzip, etc.).*

# Request and Response Multiplexing Benefits

- Interleave multiple requests and responses
  - *Requests or responses are not blocked on any other requests or response*

- A single TCP connection
  - *Multiple requests and responses can be delivered in parallel*

- Remove HTTP/1.x workarounds
  - *Concatenated files*
  - *Image sprites*
  - *Domain sharding*

- Deliver lower page load times
  - *Eliminates unnecessary latency*
  - *Improves utilization of available newtork capacity*

Overview

- Introduction
- HTTP/2
  - *Binary Framing*
  - *Stream Prioritization*
  - *Flow Control*
  - *Server Push*
  - *Header Compression*
  - *HTTP/2 Analysis*
- HTTP/3

Browser Request Prioritization

- Not all resources have equal priority when rendering a page
  - *HTTP/2 stream prioritization*
  - *Requests are prioritized based on type of asset, location on the page, and learned priority from previous visits*
    - → *If page loading was blocked on an asset, the asset priority gets increased*
- DOM, CSSOM and JavaScript
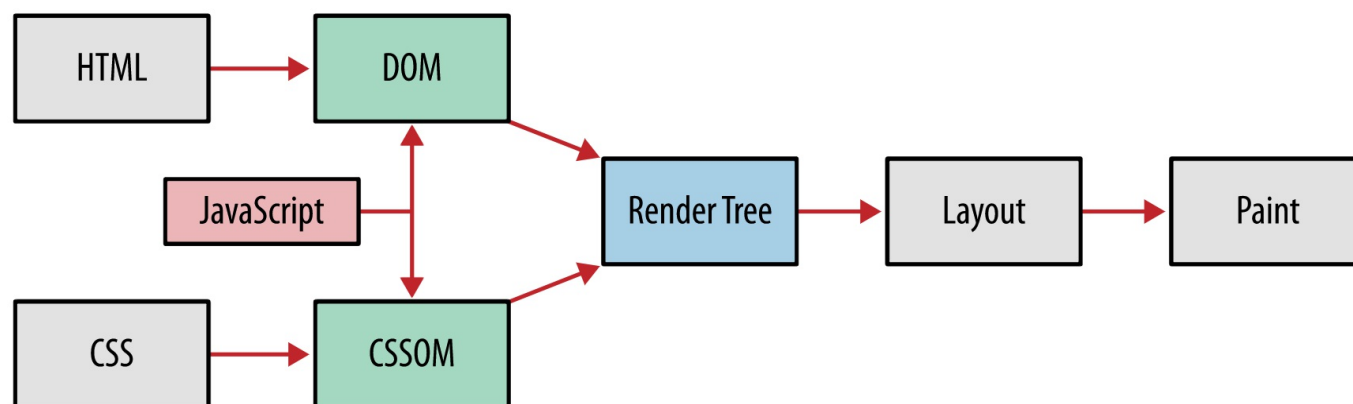
  - *HTML document is critical to construct DOM*
  - *CCS is required to construct CSSOM*
  - *Both DOM and CSSOM construction can be blocked on JavaScript resources*
    - → *A script can issue* `doc.write` *and block DOM parsing and construction*
    - → *A script can query for a computed style of an object; the script can block on CSS*

Stream Prioritization

- Purpose
  - *Messages split into **frames** which are delivered in multiplexed **streams***
  - *The order in which frames are delivered is important for a good performance*
  - *Client can define stream prioritization*
    - → *optimizations in the browser, change prioritization based on user interaction*

- Streams' weights and dependency
  - *Each stream can be assigned an integer weight between 1 and 256.*
  - *Each stream may be given an explicit dependency on another stream.*
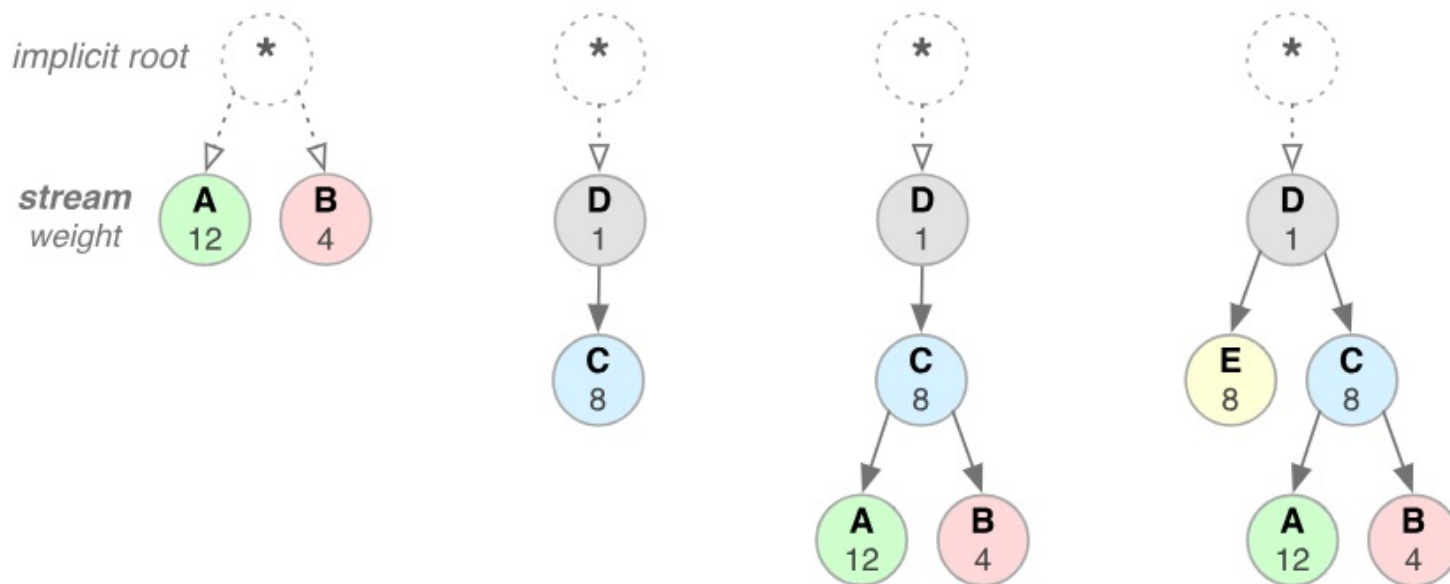
Stream Prioritization (cont.)

- Dependency
  - *referencing the unique identifier of another stream as its parent*
  - *if the identifier is omitted the stream is dependent on the "root stream"*
  - *The parent stream should be allocated resources ahead of its dependencies.*
    - → *"Please process and deliver response D before response C"*

- Weights
  - *Sibling streams have resources allocated as per their weights*
  - *Example*
    - → *Sum all the weights:* `4 + 12 = 16`
    - → *Divide each stream weight by the total weight:* `A = 12/16, B = 4/16`
    - → *Stream* `A` *receives* `3/4` *and stream* `B` *receives* `1/4` *of available resources;*

Overview

- Introduction
- HTTP/2
  - *Binary Framing*
  - *Stream Prioritization*
  - *Flow Control*
  - *Server Push*
  - *Header Compression*
 - *HTTP/2 Analysis*
- HTTP/3

Flow control

- Prevent sender from receiving data it doest not want
  - *Receiver is busy or under heavy load*
  - *Receiver is willing to allocate fixed amount of resources for a stream*
- Examples
  - *Client request a video stream; a user pauses the stream*
    - → *the client wants to pause the stream delivery to avoid buffering*
  - *A proxy server has a fast downstream and slow upstream*
    - → *the proxy server can control how quickly the downstream delivers data to match the speed of upstream*
    - → *better control of resource usage*
  - *Similar problems as in TCP flow control*
    - → *TCP flow control has no app-level API to regulate delivery of streams*
- Flow control
  - *Sender and receiver both advertise stream flow control window in bytes = the size of the available buffer space to hold the incoming data*
  - *exhanged by special* `SETTINGS` *and* `WINDOW_UPDATE` *frames*
  - *Flow control is hop-by-hop, not end-to-end*
    - → *an intermediary can set its own flow control*
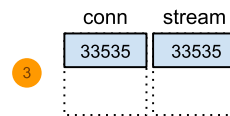
Flow Control Example

window sizes   **client**   **server**

Default window size is 65535, can be changed by SETTINGS_INITIAL_WINDOW_SIZE property in SETTINGS frame.

(1)   conn: 65535   stream: 65535

(2) DATA: 16,000 →
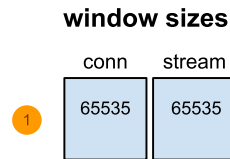
The client sends DATA frame of size 16,000 bytes and updates the window size for the stream and the connection.

(2)   conn: 49535   stream: 49535

(3) DATA: 16,000 →
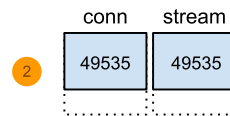
The client sends another DATA frame of size 16,000 bytes and updates the window sizes accordingly.

(3)   conn: 33535   stream: 33535

← WINDOW_UPDATE: 32,000 (stream)

(4) *The server sends WINDOW_UPDATE frame for the client to update the window sizes for the stream and the connection.*

← WINDOW_UPDATE: 32,000 (conn)

The client receives WINDOW_UPDATE and updates the window sizes accordingly.

(4)   conn: 65535   stream: 65535

- Window size
  - *The client and server maintains the window size for each stream and a connection.*
  - *How much data the client can still send to the server and vise-versa.*
  - `WINDOW_UPDATE` *provides an increment of the current window size.*
  - *When the window size is zero, no data is sent until the other party changes it.*

Overview

- Introduction
- HTTP/2
  - *Binary Framing*
  - *Stream Prioritization*
  - *Flow Control*
  - *Server Push*
  - *Header Compression*
  - *HTTP/2 Analysis*
- HTTP/3

Server push

- Ability to send multiple responses for a single request
  - *A response to the request is sent back*
  - *Additonal resources can be pushed without client requesting them*
  - *Hypertext – "server knows what the client will need"*

HTTP 2.0 connection

| stream 4 frame 1 | ... | stream 1 frame n | stream 4 *promise* | stream 2 *promise* |

| stream 1 frame 2 |

| stream 1 frame 1 |

stream 1: /page.html (client request)
stream 2: /script.js (push promise)
stream 4: /style.css (push promise)
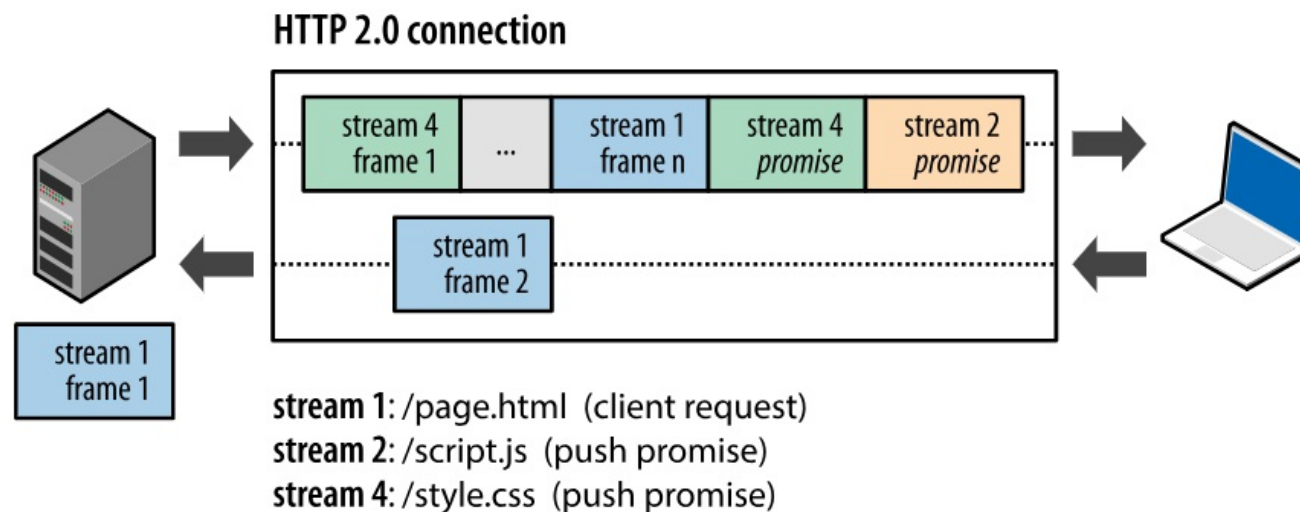
- Similar to resource inlining
  - *A resource is pushed to the client in HTML/CSS resource*
- Performance benefits
  - *Cached by the client, reused across pages, multiplexed, declined by the client*

Push promise

- `PUSH_PROMISE` frames
  - *A singnal that the server intents to push resources to the client*
  - *The client needs to know which resources the server intends to push to avoid creating duplicate requests for these resources.*

- After the client receives `PUSH_PROMISE`
  - *it may decline the stream (via `RST_STREAM` frame)*
    - → *For example, when the resource is already in the cache*
    - → *As for inline resources, this is not possible, the client always receives them*
  - *it can limit the number of concurrently pushed streams*
  - *it can adjust the initial flow control window to control how much data is pushed when the stream is first opened*
  - *it can disable server push entirely*

- pushed resources must obey the same-origin policy

Implementation

- You need to implement Server Push at the app level
  - *Your server may provide a library/API to do so*
    - → *Node express middleware:* `http2-express-autopush`
    - → *Nginx: http2 server push*

```
 1   server {
 2       # Ensure that HTTP/2 is enabled for the server
 3       listen 443 ssl http2;
 4
 5       ssl_certificate ssl/certificate.pem;
 6       ssl_certificate_key ssl/key.pem;
 7
 8       root /var/www/html;
 9
10       # whenever a client requests demo.html, also push
11       # /style.css, /image1.jpg and /image2.jpg
12       location = /demo.html {
13           http2_push /style.css;
14           http2_push /image1.jpg;
15           http2_push /image2.jpg;
16       }
17   }
```

Rules for server push

- Performance analysis by Google
  - *see* *Rules of Thumb for HTTP/2 Push*

- Server push may not always lead to a better performance, such as:
  - *Push just enough resources to fill idle network time, and no more.*

A **No Server Push**

GET html

First byte of HTML

The client reads ‹head› and a link to JS, and sends a request.

RTT

First byte of JS

The client requires one RTT to receive the first byte of JS.

HTML size   *unused capacity*

BDP

B **Server Push**
(HTML size < BDP)

GET html

First byte of HTML

<1 RTT

The client receives the first byte of JS in less than 1 RTT.

Last byte of HTML, first byte of JS

HTML size   JS size

BDP

C **Server Push**
(HTML size > BDP)

GET html

First byte of HTML

>1 RTT

Last byte of HTML, first byte of JS

The client receives the first byte of JS in more than 1 RTT.

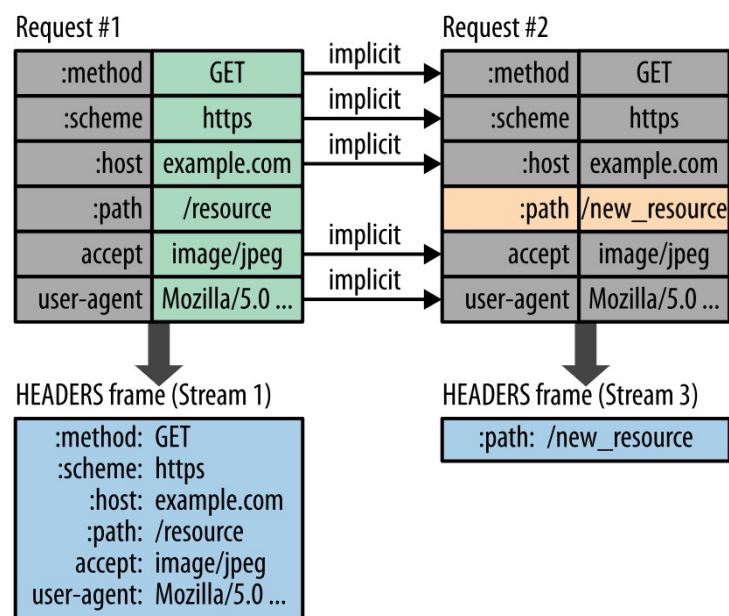HTML, more than 1 RTT to send it   JS size

BDP

- *(A) may be better than (C) when* `HTML size > BDP`
  BDP = Bandwidth-delay product

Overview

- Introduction
- HTTP/2
  - *Binary Framing*
  - *Stream Prioritization*
  - *Flow Control*
  - *Server Push*
  - *Header Compression*
  - *HTTP/2 Analysis*
- HTTP/3

Header compression

- Purpose
  - *Each HTTP request/response contains a set of headers (metadata)*
  - *HTTP/1.x – metadata sent as plain text, adds 500-800 bytes per transfer*

- HTTP/2 provides
  - *Request and response metadata are compressed using HPACK format*
    - → *header fields encoded via a static Huffman code – reduces size*
    - → *client and server maintain an **indexed list of previously seen header fields** in static and dynamic tables*

Request #1

:method	GET
:scheme	https
:host	example.com
:path	/resource
accept	image/jpeg
user-agent	Mozilla/5.0 ...

implicit
implicit
implicit

implicit
implicit

Request #2

:method	GET
:scheme	https
:host	example.com
:path	/new_resource
accept	image/jpeg
user-agent	Mozilla/5.0 ...

HEADERS frame (Stream 1)

:method: GET
:scheme: https
:host: example.com
:path: /resource
accept: image/jpeg
user-agent: Mozilla/5.0 ...

HEADERS frame (Stream 3)

:path: /new_resource

Static and Dynamic Tables

- Static table
  - *Pre-defined table of* `61` *header fiedls and values where each has assigned an index*
  - *Defined in HPACK: Header Compression for HTTP/2*

```
GET header field in HEADERS frame

Header: :method: GET
 Name Length: 7
 Name: :method
 Value Length: 3
 Value: GET
 :method: GET
 [Unescaped: GET]
 Representation: Indexed Header Field
 Index: 2
```

```
Static table
+--------+------------------------------------+----------------+
| Index  | Header Name                        | Header Value   |
+--------+------------------------------------+----------------+
| 1      | :authority                         |                |
| 2      | :method                            | GET            |
| 3      | :method                            | POST           |
| 4      | :path                              | /              |
| 5      | :path                              | /index.html    |
```

  - *Decompressed size is* `10` *bytes (* `name+value` *length)*
  - *The real size is* `1` *byte.*

- Dynamic table
  - *Dynamically created table of header fields that do not exist in the static table.*
  - *It is maintained by the client and the server*
  - *There are entries starting with index* `62`

Overview

- Introduction
- HTTP/2
  - *Binary Framing*
  - *Stream Prioritization*
  - *Flow Control*
  - *Server Push*
  - *Header Compression*
  - *HTTP/2 Analysis*
- HTTP/3

nghttp

- Command line tool
  - *nghttp displays frames and a summary information about each frame*

```
$ nghttp -vvv -n https://w20.vitvar.com/lecture1.html
[  0.071] Connected
The negotiated protocol: h2
[  0.137] send SETTINGS frame <length=12, flags="0x00," stream_id="0">
          (niv=2)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
[  0.137] send PRIORITY frame <length=5, flags="0x00," stream_id="3">
          (dep_stream_id=0, weight=201, exclusive=0)
[  0.137] send PRIORITY frame <length=5, flags="0x00," stream_id="5">
          (dep_stream_id=0, weight=101, exclusive=0)
[  0.137] send PRIORITY frame <length=5, flags="0x00," stream_id="7">
          (dep_stream_id=0, weight=1, exclusive=0)
[  0.137] send PRIORITY frame <length=5, flags="0x00," stream_id="9">
          (dep_stream_id=7, weight=1, exclusive=0)
[  0.137] send PRIORITY frame <length=5, flags="0x00," stream_id="11">
          (dep_stream_id=3, weight=1, exclusive=0)
[  0.137] send HEADERS frame <length=49, flags="0x25," stream_id="13">
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=11, weight=16, exclusive=0)
          ; Open new stream
          :method: GET
          :path: /lecture1.html
          :scheme: https
          :authority: w20.vitvar.com
          accept: */*
          accept-encoding: gzip, deflate
          user-agent: nghttp2/1.42.0
[  0.180] recv SETTINGS frame <length=6, flags="0x00," stream_id="0">
          (niv=1)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
[  0.180] recv WINDOW_UPDATE frame <length=4, flags="0x00," stream_id="0">
          (window_size_increment=16711681)
[  0.180] recv SETTINGS frame <length=0, flags="0x01," stream_id="0">
          ; ACK
          (niv=0)
[  0.180] recv (stream_id=13) :status: 200
[  0.180] recv (stream_id=13) server: GitHub.com
...
```
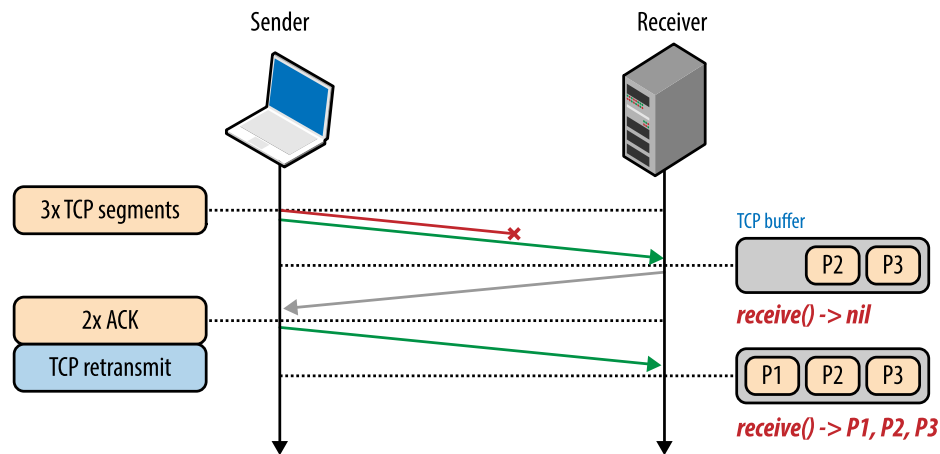
Wireshark and HTTP/2 Traffic

- HTTP/2 is encrypted
  - *You can capture packets in Wireshark/tcpdump but they are encrypted*
  - *The browser (FF, Chrome, Opera) can dump keys in NSS Key Log Format*
  - *You can use the log to decrypt the communication in Wireshark*

- Demo at `https://kde.vitvar.com`

  1. *Configure an env variable* `SSLKEYLOGFILE` *to point to a file on the filesystem*
  2. *Start **firefox** browser; check the keylog file was created.*
  3. *Start **wireshark**; configure the keylog file in pre-master secret log filename*
  4. *Start packet capture on* `eth0` *using the filter:*

     `((dst host 185.199 and src host 192.168) or`

     `(dst host 192.168 and src host 185.199))`

     - *This captures the packets in both directions between the client and the server and back*
  5. *Point **firefox** to https://vitvar.com*
  6. *Check captured packets in the **wireshark***
     - *There should be decrypted HTTP/2 communication.*

Overview
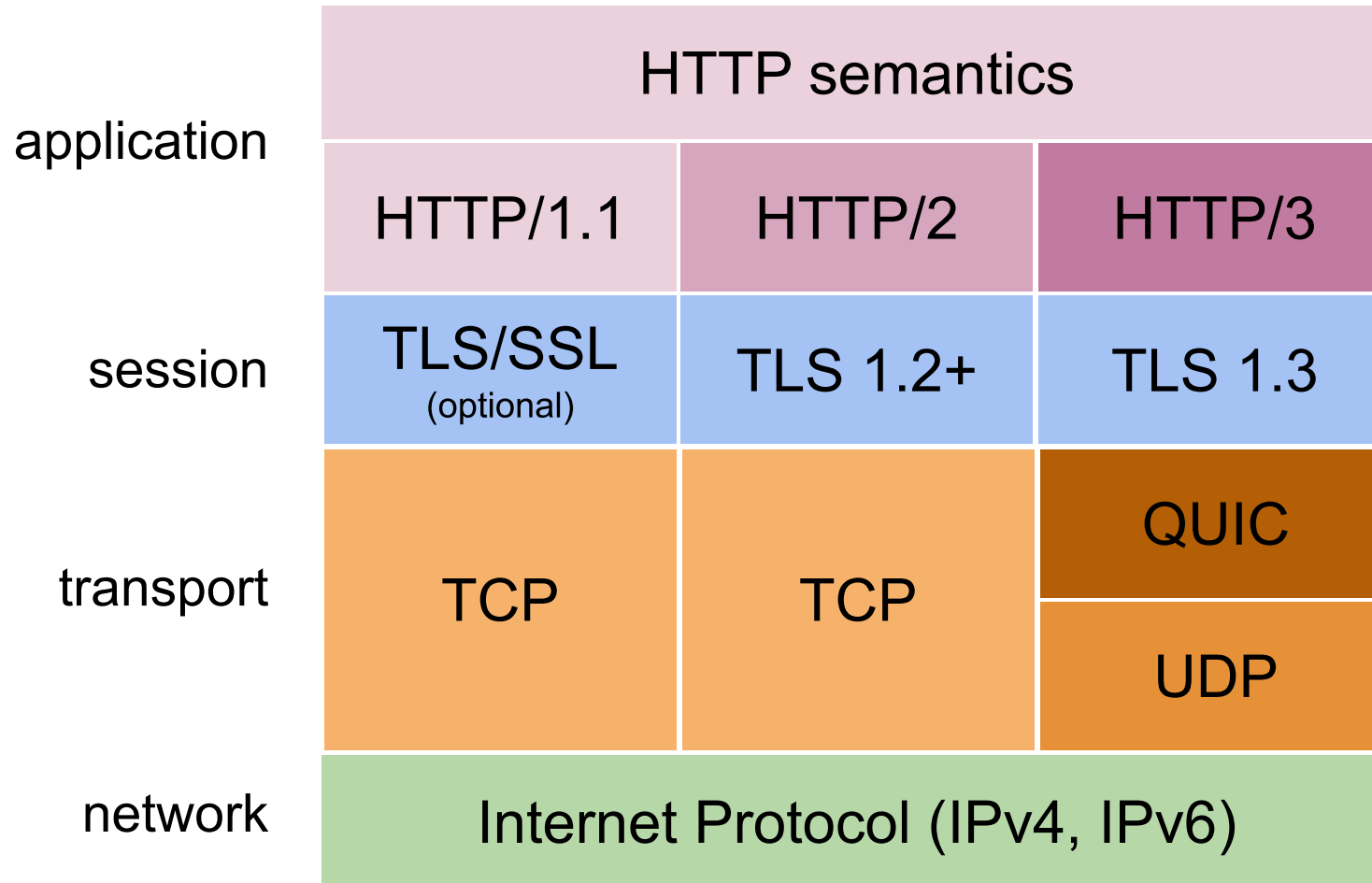
- Introduction
- HTTP/2
- HTTP/3

HTTP/2 Drawbacks

- HTTP/2 is dependant on TCP

- TCP head-of-line blocking

Sender   Receiver

3x TCP segments

TCP buffer

| P2 | P3 |

receive() -> nil

2x ACK

TCP retransmit
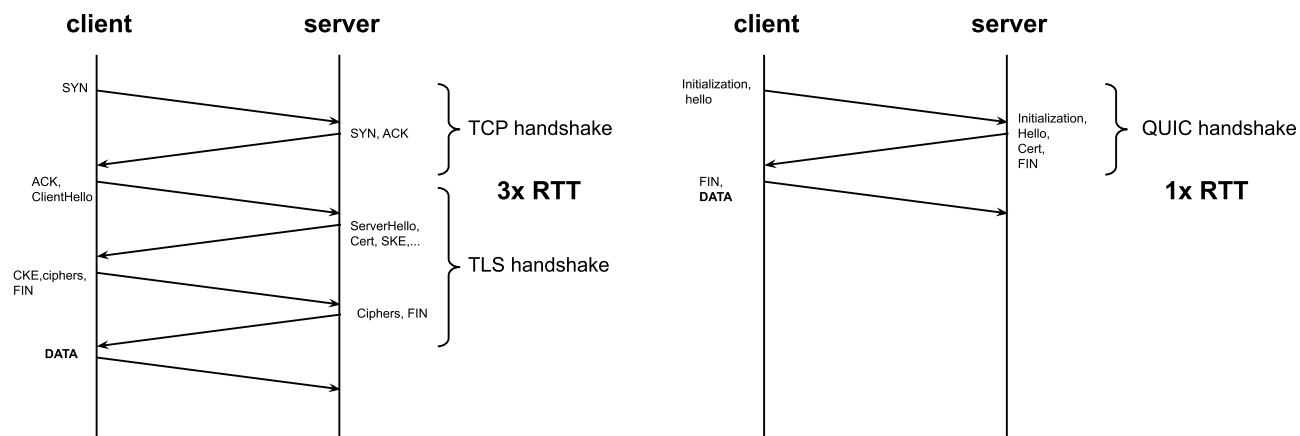
| P1 | P2 | P3 |

receive() -> P1, P2, P3

     – *When TCP segment does not arrive, it needs to be transmitted*
     – *This may delay all HTTP/2 streams*

- There must always be TLS handshake after TCP hanshake
  – *HTTP/2 can only be used with TLS*

HTTP/3

- Protocol stack

	application			session			transport			network

	HTTP semantics	
HTTP/1.1	HTTP/2	HTTP/3
TLS/SSL (optional)	TLS 1.2+	TLS 1.3
TCP	TCP	QUIC / UDP
Internet Protocol (IPv4, IPv6)		

Connection Establishment

- # Reduces overhead during connection setup
  - ## *H2 requires TLS, thus requires 3 RTT before data is sent*
  - ## *QUIC requires 1 RTT before data is sent*

Recovery

- TCP
  - *TCP performs loss recovery (retransmition)*
  - *Multiple frames in a single TCP segment, when TCP segment is lost, the whole segment (with all frames) need to be retransmitted.*

- QUIC
  - *QUIC implements recovery at the stream level*
  - *Lost or corrupted data within a specific stream can be recovered independently without affecting other streams.*
  - *Forward Error Correction (FEC): redundant data to allow the receiver to recover lost packets without the need for retransmission.*

Networks Switching

- A user moves from one network to another.

- TCP
  - *Connections timeout; they are re-established*
  - *This is a lengthy process*

- QUIC
  - *QUIC includes unique connection ID with the server*
  - *Connection ID is independent of the source*
  - *Pakcets are re-send with the same IDs*
    - *→ they are still valid when the source changes*