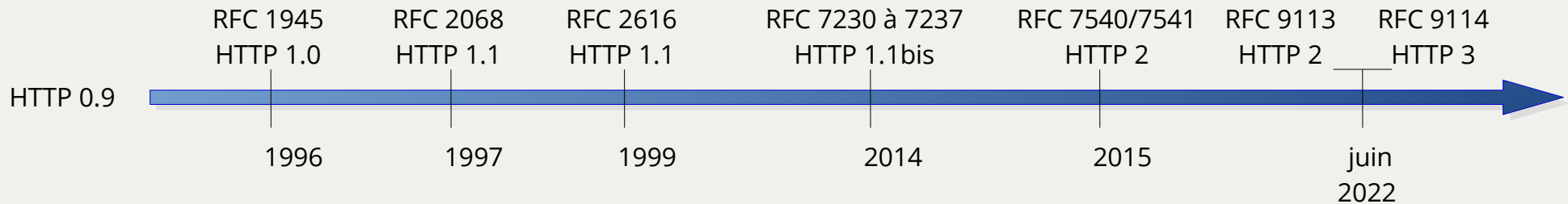


Le protocole applicatif HTTP

Le protocole HTTP

- HTTP: *Hypertext Transfert Protocol*
 - Hypertexte : un document ou (un ensemble de documents) contenant des informations liées entre elles par des hyperliens.
- Protocole de niveau applicatif selon le modèle OSI
- Développement initié par Tim Berners-Lee au CERN en 1989
- HTTP est la base des communications du *World Wide Web*
- Les RFCs définissant les versions successives d'HTTP :

Application
Présentation
Session
Transport
Réseau
Liaison de données
Physique



- Protocole sans état fonctionnant sur le schéma requête/réponse

HTTP/0.9

- La version HTTP/0.9 est définie en 1991
- Protocole simple:
 - connexion du client HTTP
 - envoi d'une requête (GET)
 - réponse du serveur HTTP
 - fin de la réponse, fermeture de la connexion

HTTP/1

HTTP/1.0

- La version HTTP/1.0 est définie par la RFC 1945 en 1996
- HTTP/1.0 introduit de nouveaux en-têtes pour assurer le transfert de méta-données
 - Host : précise le site Web concerné par la requête
 - nécessaire pour un serveur hébergeant plusieurs sites à la même adresse IP
 - Referer : Indique l'URI de la page qui a donné un lien sur la ressource demandée
 - Permet de connaître d'où viennent les visiteurs d'un site
 - User-Agent : Indique le logiciel utilisé pour se connecter
 - Généralement un navigateur web ou d'un robot d'indexation, mais aussi des outils en ligne de commande comme curl
- Supporte de nouvelles méthodes : HEAD et POST
- Dans HTTP/1.0, 1 requête/réponse = 1 connexion TCP
 - Connexion fermée après réception de la réponse

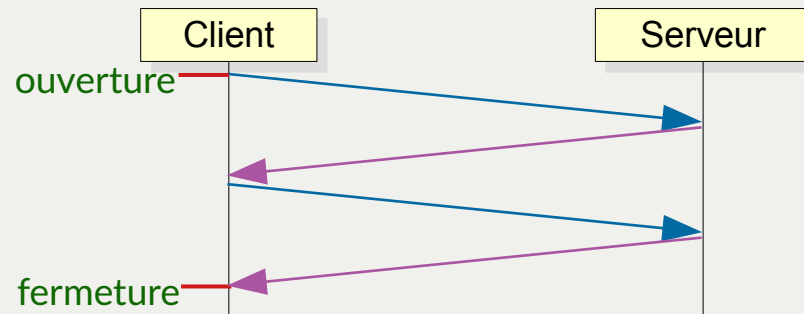
HTTP/1.1

- La version HTTP/1.1 est définie en
 - 1997 par la RFC 2068 (rendue obsolète par la RFC 2616)
 - 1999 par la RFC 2060 (rendue obsolète par les RFC 7230 à 7235)
 - 2014 par les RFC 7230 à 7235 (rendues obsolètes par les RFC 9110 et 9112)
 - Juin 2022 les RFC 9110 et 9112
- HTTP/1.1 améliore la gestion du cache, rend obligatoire l'en-tête Host dans les requêtes
- L'en-tête `Connection:Keep-Alive` permet des connexions persistantes
 - Plusieurs requêtes peuvent être émises au sein de la même connexion (*pipeline*)
 - Moins de latence dans les transferts (plus de perte de temps liée à l'ouverture de nouvelles connexions)

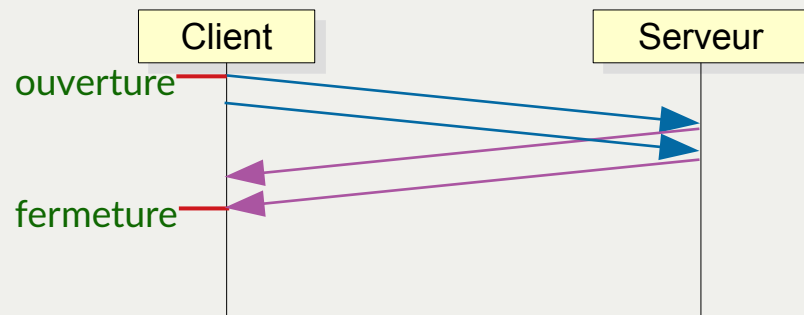
HTTP/1.0 vs HTTP/1.1

- HTTP/1.0
 - 1 requête/réponse = 1 connexion TCP
 - Pas de pipeline
- HTTP/1.1 :
 - 1 connexion TCP persistante
 - Pipeline
 - Plusieurs requêtes peuvent être émises à la suite sans attendre leurs résultats (file)
 - Les réponses sont ordonnées en fonction des requêtes (file)

- Sans pipeline



- Avec pipeline









Limites d'HTTP/1.x









- Les requêtes et les réponses sont mises en file au sein d'une connexion
 - Une réponse peut bloquer une autre
 - L'utilisation de la connexion TCP n'est pas optimale
 - Pour améliorer les performances (i.e. télécharger les données plus rapidement) des navigateurs Web peuvent ouvrir plusieurs connexions

HTTP/1.0 vs HTTP/1.1

- HTTP 1.0

Name	Meth...	Status	Protocol	Sche...	Remote A...	Type	Initiator	Size	Time	Co...	Waterfall	▲
 0.0.0.0	GET	200	http/1.0	http	0.0.0.0:80...	docu...	Other	496 B	5 ms	4384		
 engine.io.js	GET	200	spdy	https	104.19.19...	script	(index)	27.2 ...	90 ms	4400		
 bundle.js	GET	200	http/1.0	http	0.0.0.0:80...	script	(index)	66.6 ...	10 ms	4399		

- HTTP 1.1

Name	Method	Status	Protocol	Remote Address	Initiator	Conne...	Waterfall	▲
 empty.css	GET	200	http/1.1	130.89.148.14:443	(index)	4218		
 identica.png	GET	200	http/1.1	130.89.148.14:443	(index)	4218		
 planet.png	GET	200	http/1.1	130.89.148.14:443	(index)	4242		
 gradient.png	GET	200	http/1.1	130.89.148.14:443	(index)	4241		

Les messages HTTP/1.x

- Structure des en-têtes des messages HTTP :

- ```
message-header = field-name ":" field-value
general-header = Cache-Control | Connection | Date | Pragma | Trailer |
 Transfer-Encoding | Upgrade | Via | Warning

entity-header = Allow | Content-Encoding | Content-Language |
 Content-Length | Content-Location | Content-Type | Expires |
 ...
```

- Corps d'un message

```
message-body = entity-body
entity-body = OCTET*
```

<https://tools.ietf.org/html/rfc2616>

# Format des requêtes HTTP/1.x

```
Request = Request-Line ((general-header | request-header | entity-header) CRLF)*
 CRLF [message-body]
```

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

```
Method = GET | HEAD | POST | PUT | DELETE | OPTIONS | CONNECT
```

```
request-header = Accept | Accept-Charset | Accept-Encoding | Accept-Language |
 Authorization | Expect | From | Host | User-Agent | ...
```

# Méthodes HTTP

|         |                                                                                       |
|---------|---------------------------------------------------------------------------------------|
| TRACE   | requête de diagnostic pour connaître le cheminement à travers les proxys (nilpotente) |
| GET     | pour récupérer une ressource, méthode la plus employée (nilpotente)                   |
| POST    | pour envoyer des données vers une ressource                                           |
| DELETE  | pour supprimer une ressource du serveur (idempotente)                                 |
| PUT     | pour enregistrer une ressource sur le serveur (idempotente)                           |
| HEAD    | pour obtenir les en-têtes d'une ressource, pas le corps (nilpotente)                  |
| OPTIONS | pour obtenir les capacités du serveur HTTP (nilpotente)                               |
| CONNECT | pour ouvrir un tunnel afin d'envoyer des données brutes                               |

**Nilpotence** : pas d'effet de bord sur le serveur

**Idempotence** : la réexécution d'une même requête n'a pas d'incidence

# Exemple d'utilisation de la méthode HTTP GET

```
GET http://localhost/commande.php?user=123&action=Valider HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7)
 Gecko/20040808 Firefox/0.9.3
Accept: text/xml,application/xml,application/xhtml+xml,
 text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: fr-fr,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-15,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
```

# Exemple d'utilisation de la méthode HTTP POST

```
POST http://localhost/commande.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7)
 Gecko/20040808 Firefox/0.9.3
Accept: text/xml,application/xml,application/xhtml+xml, text/html;q=0.9,
 text/plain;q=0.8,image/png,*/*;Accept-Language: fr-fr,en-us;q=0.7,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-15,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Content-Type: text/plain
Content-Length: 23
user=123
action=Valider
```

# Format des réponses HTTP/1.x

```
Response = Status-Line ((general-header | response-header | entity-header) CRLF)*
CRLF
[message-body]
```

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

```
response-header = Accept-Ranges | Age | Location | Proxy-Authenticate
 | Server | WWW-Authenticate |...
```

# Codes d'état des réponses HTTP/1.x

| Codes | Type <i>Reason-Phrase</i> | Exemples                                                                     |
|-------|---------------------------|------------------------------------------------------------------------------|
| 1xx   | Information               |                                                                              |
| 2xx   | Succès                    | 200 ( <i>Ok</i> , Requête traitée avec succès)                               |
| 3xx   | Redirection               | 302 ( <i>Found</i> , Document déplacé de façon temporaire)                   |
| 4xx   | Erreur côté client        | 400( <i>Bad Request</i> ), 403 ( <i>Forbidden</i> ), 404( <i>Not Found</i> ) |
| 5xx   | Erreur côté serveur       | 500 ( <i>Internal Error</i> ), 503 ( <i>Service Unavailable</i> )            |



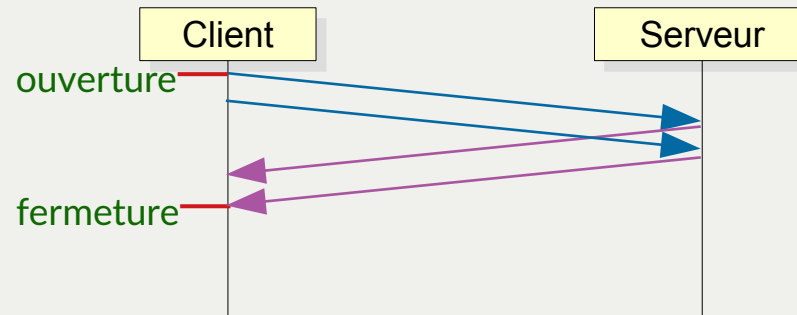
# Exemple de réponse HTTP/1.x

```
HTTP/1.1 200 OK
Date: Tue, 26 Jun 2018 14:44:13 GMT
Server: Apache/2.4.18 (Ubuntu)
Last-Modified: Tue, 26 Jun 2018 13:48:28 GMT
Etag: "2d719-56f8bc0bf3f5f-gzip"
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Encoding: gzip
Cache-Control: max-age=1
Expires: Tue, 26 Jun 2018 14:44:14 GMT
Content-Length: 31769
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
Content-Language: fr

<!DOCTYPE html PUBLIC ...
```

# Limitation d'HTTP/1.X

- En-têtes verbeux
- Utilisation non optimale des capacités du réseau et des connexions TCP
  - Réception séquentielle au sein d'une connexion TCP
    - Ouverture de plusieurs connexions TCP pour paralléliser le chargement des ressources



# HTTP/2

# Présentation d'HTTP/2

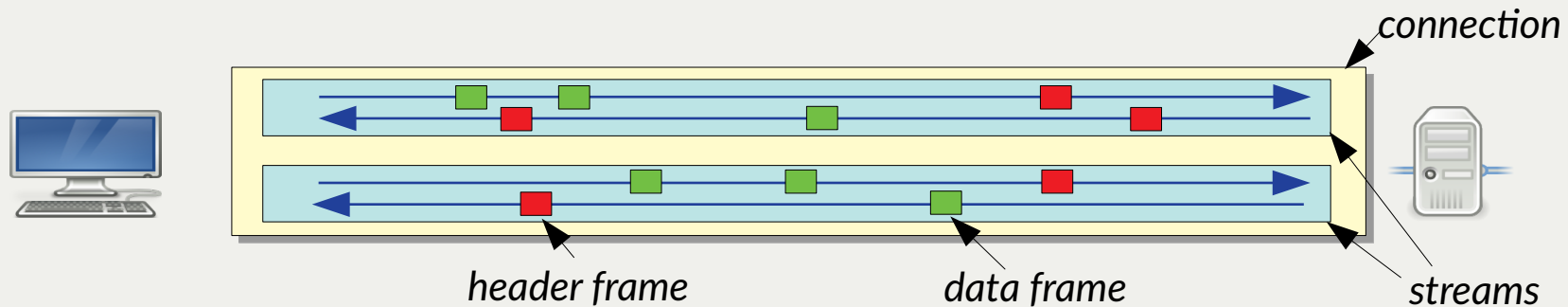
- Protocole applicatif reposant sur TCP et définit par la RFC 9113 (de juin 2022), qui rend la RFC 7540 (de mai 2015) obsolète
- Compatible avec HTTP/1.1
- HTTP/2.0 apporte des améliorations de performances en
  - reposant sur des « *frames* » au format binaire
  - introduisant la notion de flux et un système de multiplexage
  - implantant une compression des en-têtes
  - proposant un mode « *push server* »
  - permettant une priorisation des requêtes
- Une seule connexion
  - Meilleure gestion de la congestion entre les nœuds

# Identification et établissement des connexions

- HTTP/2 sur TLS (*Transport Layer Security*) est identifié par la chaîne « h2 »
  - « h2 » sérialisée dans l'identifiant de protocole dans l'extension TLS-ALPN (*TLS Application-Layer Protocol Negotiation*)
- Dans la version de 2015, la chaîne « h2c » identifie HTTP/2 sur une connexion TCP textuelle
  - Utilisée avec le mécanisme d'HTTP/1.1 *Upgrade* pour faire une négociation de protocole
  - Dépréciée dans version de juin 2022 d'HTTP/2
- Pour créer une connexion HTTP/2, un client émet une requête HTTPS (avec TLS)
- Quand la négociation TLS est complète, le client et le serveur émettent une préface de connexion pour confirmer l'utilisation du protocole et établir les paramètres de la connexion HTTP/2
  - Frame SETTINGS

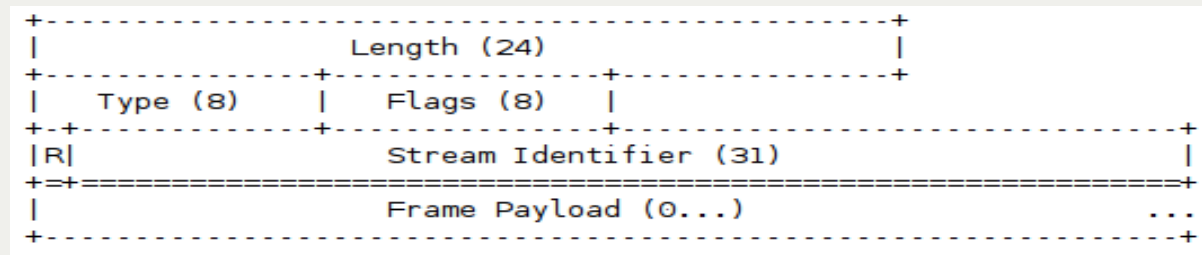
# Connexion, flux et multiplexage

- Une connexion peut contenir plusieurs flux ouverts de manière concurrente
- Flux
  - canal de communication virtuel
  - séquence bidirectionnelle de *frames* échangées entre un client et un serveur
    - L'ordre des *frames* dans un flux a une importance
- Les flux peuvent être établis et utilisés unilatéralement ou être partagés entre le client et le serveur



# Format des *frames* HTTP2

- En-tête de 9 octets pour toutes les *frames*
  - Taille du *payload* (entier sur 24 bits)
  - Type de *frame* (sur 8 bits)
  - *Flags* (8 bits pour valeurs booléennes spécifiques au type de *frame*)
  - R (1 bit réservé, dont la sémantique n'est pas définie)
  - Identifiant de flux (sur 31 bits)
  - La charge utile (*payload*)



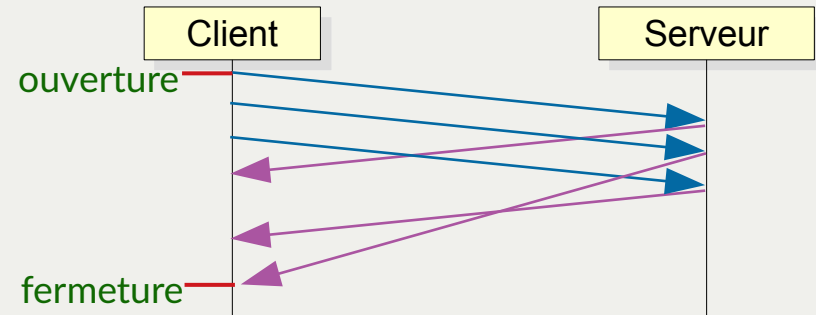
# Types de frames

HEADERS	Frames utilisées pour ouvrir des flux. Elles peuvent contenir un bloc d'en-têtes
DATA	Frames contenant un nombre arbitraire d'octets, et transportant les charges utiles des requêtes et des réponses
PRIORITY	Frames précisant la priorité du flux du point de vue de l'émetteur
RST_STREAM	Frames émises pour demander la fermeture d'un flux
SETTINGS	Frames définissant des paramètres de configuration qui peuvent influencer le client ou le serveur
PING	Frames utilisées pour mesurer le temps d'aller/retour entre un client et un serveur
GO_AWAY	Frames émises pour mettre un terme aux connexions
PUSH_PROMISE	Frames avertissant le client que le serveur « pousse » des données de lui même





































# HTTP 2 : Multiplexage et priorisation des flux

- Multiplexage des flux
  - Permet de résoudre les limitations de HTTP 1.x et de mieux exploiter une connexion TCP
    - Les réponses ne sont pas bloquées en attente (moins de latence)
  - Meilleure gestion de la congestion entre le client et le serveur



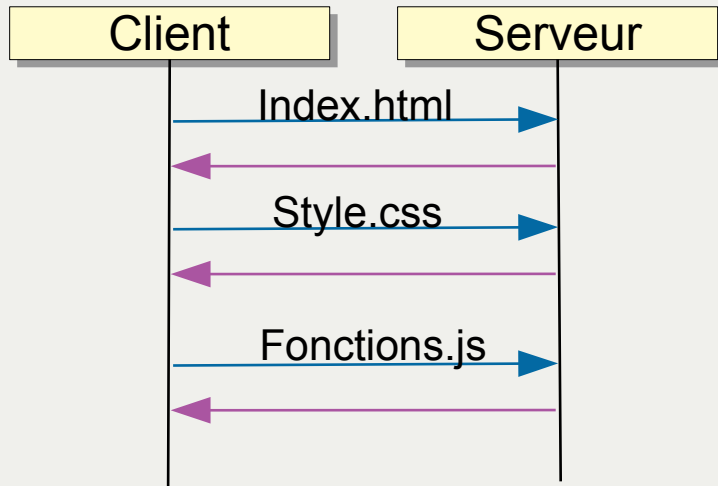
- Priorisation des flux
  - Un poids (entre 1 et 256) est associé à chaque flux par le client pour indiquer au serveur quelles réponses il souhaite obtenir en priorité
  - Le serveur peut gérer ses ressources (mémoire, CPU et réseau) pour répondre au mieux au client en fonction des priorités

# HTTP 2 : Multiplexage et priorisation des flux

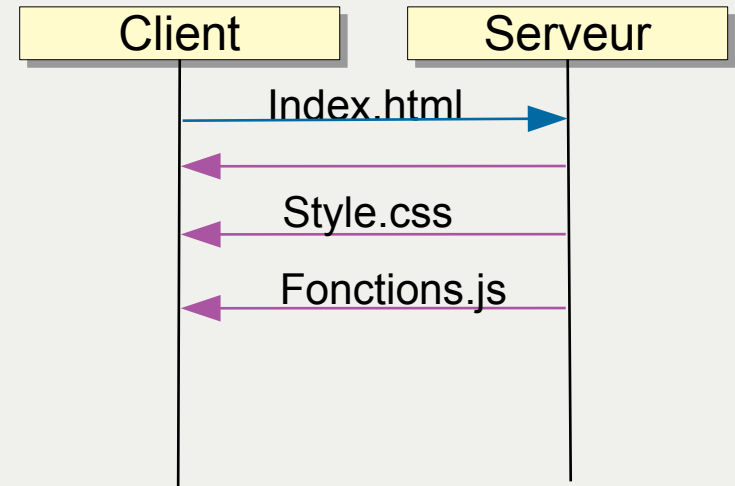
Name	Method	Status	Protocol	Remote Address	Initiator	Connection ID	Waterfall
 fetch.js?bust=6629d03	GET	200	h2	23.200.152.69:443	require.js?bus...	349	
 fonticon.woff2?bust=	GET	200	h2	23.200.152.69:443	(index)	463	
 Canaro-SemiBold.woff2	GET	200	h2	23.200.152.69:443	(index)	463	
 oMMgfZMQthOryQo9n22dcuvvDin1pK8...	GET	200	spdy	216.58.201.227:443	(index)	426	
 d-6IYpIOFocCacKzxwXSOJBw1xU1rKpt...	GET	200	spdy	216.58.201.227:443	(index)	426	
 Canaro-Light.woff2	GET	200	h2	23.200.152.69:443	(index)	463	
 Canaro-ExtraBold.woff2	GET	200	h2	23.200.152.69:443	(index)	463	
 fontawesome-webfont.woff2?v=4.7.0	GET	200	h2	23.200.152.69:443	(index)	463	
 ua-parser.js?bust=6629d03	GET	200	h2	23.200.152.69:443	require.js?bus...	349	
 antvoice.js?bust=6629d03	GET	200	h2	23.200.152.69:443	require.js?bus...	349	
 burger-menu.js?bust=6629d03	GET	200	h2	23.200.152.69:443	require.js?bus...	349	
 cookie.js?bust=6629d03	GET	200	h2	23.200.152.69:443	require.js?bus...	349	
 date-formatter.js?bust=6629d03	GET	200	h2	23.200.152.69:443	require.js?bus...	349	
 duration-formatter.js?bust=6629d03	GET	200	h2	23.200.152.69:443	require.js?bus...	349	
 dmp.js?bust=6629d03	GET	200	h2	23.200.152.69:443	require.js?bus...	349	
 dmp-api.js?bust=6629d03	GET	200	h2	23.200.152.69:443	require.js?bus...	349	
 estat.js?bust=6629d03	GET	200	h2	23.200.152.69:443	require.js?bus...	349	

# Push Server

- Une page HTML contient généralement du texte mais aussi des inclusions à des feuilles de style, des fichiers JS ou des liens vers des images.



*sans push server*



*avec push server*

# HPACK : Compression des en-têtes HTTP/2

- Les en-têtes HTTP/2 sont définies comme des couples clé/valeur
- La compression des en-têtes HTTP/2 est définie par la RFC 7541
  - Codage de Huffman des en-têtes
  - Codage différentiel
- Exemple :

HTTP/1.1

```
GET /index.html HTTP/1.1
Host: www.host.com
Referer:https://www.host.com/
Accept-Encoding:gzip
```

Requête 1

```
GET /logo.svg HTTP/1.1
Host: www.host.com
Referer:https://www.host.com/index.html
Accept-Encoding:gzip
```

Requête 2

```
:method: GET
:scheme: https
:host: www.host.com
:path: /index.html
referer: https://www.host.com/
accept-encoding: gzip
```

Requête 1

```
:path: /logo.svg
referer: https://www.host.com/index.html
```

Requête 2

HTTP/2

# Exemple de requête simple

```
GET /resource HTTP/1.1
```

```
Host: example.org
```

```
Accept: image/jpeg
```

```
HEADERS
```

```
==>
```

```
+ END_STREAM
```

```
+ END_HEADERS
```

```
:method = GET
```

```
:scheme = https
```

```
:authority = example.org
```

```
:path = /resource
```

```
host = example.org
```

```
accept = image/jpeg
```

<https://tools.ietf.org/html/rfc9113>

# Exemple de réponse simple

```
HTTP/1.1 304 Not Modified HEADERS
 ETag: "xyzzzy" ==> + END_STREAM
 Expires: Thu, 23 Jan ... + END_HEADERS
 :status = 304
 etag = "xyzzzy"
 expires = Thu, 23 Jan ...
```

<https://tools.ietf.org/html/rfc9113>

# Exemple de requête complexe

```
POST /resource HTTP/1.1
Host: example.org
Content-Type: image/jpeg
Content-Length: 123

{binary data}
```

==>

```
HEADERS
- END_STREAM
- END_HEADERS
:method = POST
:authority = example.org
:path = /resource
:scheme = https

CONTINUATION
+ END_HEADERS
content-type = image/jpeg
host = example.org
content-length = 123

DATA
+ END_STREAM
{binary data}
```

<https://tools.ietf.org/html/rfc9113>

# Exemple de réponse avec un contenu

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 123
{binary data}
```

==>

```
HEADERS
- END_STREAM
+ END_HEADERS
:status = 200
content-type = image/jpeg
content-length = 123

DATA
+ END_STREAM
{binary data}
```

<https://tools.ietf.org/html/rfc9113>



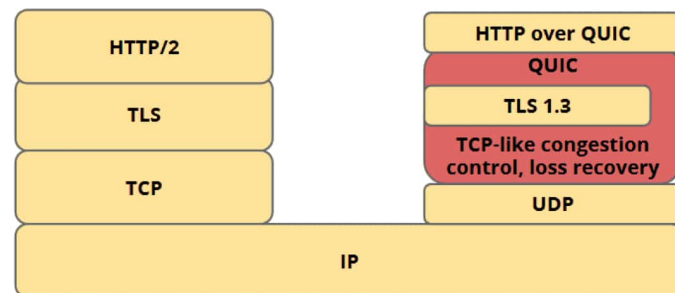
# Limitations d'HTTP/2

- HTTP/2 implante un multiplexage de flux sur une connexion TCP
- La perte d'un paquet dans un flux entraîne le blocage de tous les flux
  - TCP transmet les paquets de données de manière chronologique et de manière fiable (accusé de réception)
  - Attente de retransmission du paquet perdu
  - Problème connu sous le nom de « *Head-of-line Blocking* »

# HTTP/3

# Présentation d'HTTP/3

- HTTP/3 est défini par la RFC 9114 de juin 2022
- Les messages d'HTTP/3 sont similaires à ceux d'HTTP/2 (formats, types, sémantiques)
- HTTP/3 intègre une compression des en-têtes à l'aide de QPACK
  - QPACK est similaire à la compression HPACK d'HTTP/2
- HTTP/3 repose sur les protocoles UDP et QUIC, et non sur TCP
  - La gestion des flux diffère entre HTTP/2 et HTTP/3



# Présentation de QUIC

- QUIC est une proposition de protocole, formulée pour la première fois 2012, par Google
- QUIC a été standardisé par l'IETF en mai 2021 par la RFC 9000
- QUIC s'appuie sur UDP
- QUIC intègre nativement un mécanisme de chiffrement (TLS 1.3)
  - Le protocole TLS est donc obligatoire pour HTTP/3
  - vise à empêcher les équipements intermédiaires d'altérer ou d'écouter le trafic.
- QUIC introduit des mécanismes de multiplexage et de gestion de flux et de fiabilisation des transmissions (retransmission des paquets perdus)
- QUIC redéfinit l'établissement des contacts (*handshake*)
  - QUIC peut envoyer d'un seul coup à un serveur les requêtes de connexion et de chiffrement ce qui réduit de moitié la latence des nouvelles connexions et permet d'avoir une latence nulle pour les connexions déjà connues

# Présentation de QUIC

- Avec QUIC, les flux sont indépendants
  - La perte de paquets dans un flux, n'entraîne pas le blocage des autres flux
- QUIC assure que les données arrivent dans l'ordre dans les flux, mais pas l'ordre entre les flux
- Chaque flux établi possède un identifiant unique
- Ces identifiants persistent à travers les changements d'adresse IP
  - Ils peuvent aider à sécuriser les téléchargements ininterrompus sur, par exemple, un passage de la 4G vers le WiFi.
- Les connexions QUIC bénéficient également de l'élimination de la surcharge du handshake TCP, ce qui réduit la latence

# HTTP/2 vs HTTP/3

- Différences :
  - Le HTTP/3 utilise QUIC/UDP contrairement au HTTP/2 qui utilise TCP.
  - Grâce au chiffrement intégré TLS 1.3, le HTTP/3 évite une requête de chiffrement supplémentaire (handshake) au niveau TLS et supprime ainsi les requêtes de sécurité superflues.
  - Contrairement au HTTP/2, le HTTP/3 supporte uniquement les connexions chiffrées en raison du chiffrement intégré TLS 1.3.

# HTTP/2 vs HTTP/3

- Similarités :
  - Les deux protocoles utilisent la compression d'en-tête.
    - Le HTTP/3, cependant, remplace la compression HPACK HTTP/2 liée à une séquence de paquets par QPACK.
  - Comme le HTTP/2, le HTTP/3 prend en charge le Server Push
  - Les deux protocoles utilisent le multiplexage requête-réponse, c'est-à-dire les flux parallèles de données de différentes ressources.
  - La priorisation des flux dans les deux protocoles assure que le contenu de la page est chargé en priorité sans attendre la notification de réussite des autres requêtes.

# Bibliographie

- RFC
  - <https://tools.ietf.org/html/rfc1945>
  - <https://tools.ietf.org/html/rfc2616>
  - <https://tools.ietf.org/html/rfc7230>
  - <https://tools.ietf.org/html/rfc7540>
  - <https://tools.ietf.org/html/rfc7541>
  - <https://tools.ietf.org/html/rfc9113>
  - <https://tools.ietf.org/html/rfc9114>