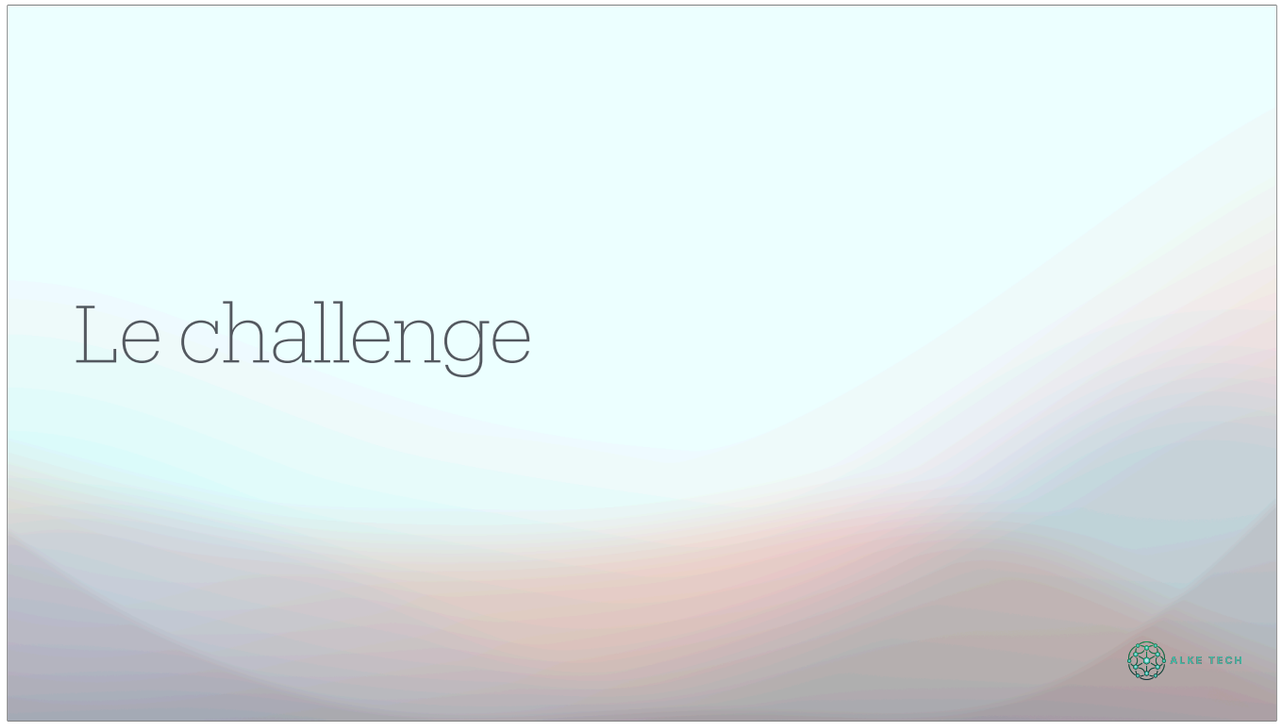
180 000 requêtes par seconde

Expliqué simplement

Xavier Leune

ALKE TECH



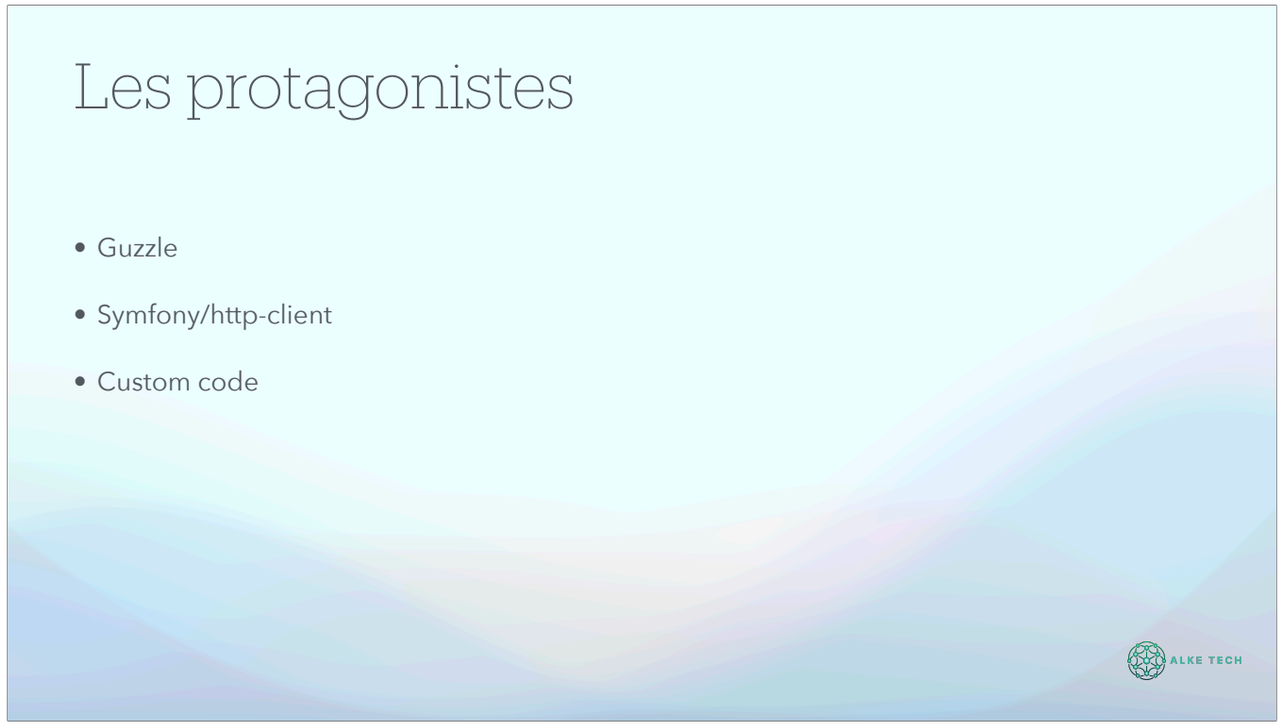
Aujourd’hui je vais vous raconter comment j’ai eu à résoudre un challenge: émettre un maximum de requêtes simultanées vers un nombre limité d’origines avec pour but d’envoyer des notifications webpush avec une vitesse contrôlée. Pour cela j’ai passé en revue les options qui s’offraient à moi et je me suis demandé jusqu’à quel point je pourrais pousser les limites de la parallélisation des requêtes. Nous allons donc revoir ensemble comment passer à l’échelle ces requêtes.

Je m'appelle Xavier Leune, j'ai 18 ans d'expérience en PHP et je suis fondateur d'Alke Tech, où je construit le futur des outils d'analytics.

Les conditions du test

- Serveur distant
- PHP 8.4
- Poste local + serveur tunés (paramètres réseau du kernel)

ALKE TECH



La première question que je me suis posé était donc de savoir si une solution existante me permettrait de répondre à mes contraintes pour effectuer les requêtes:
- Pouvoir gérer finement le nombre de requêtes que j'envoie par seconde afin d'obtenir une vitesse quasi-constante
- Pouvoir exécuter des traitements en dehors de la boucle de lecture des réponses (et donc une lecture non bloquante)
- Pouvoir scaler progressivement le nombre de requêtes effectuées (ramp-up)

Guzzle et HttpClient sont 2 supers options comme clients http, très polyvalents avec un très bon support des requêtes en // - je me sers la plupart du temps d'une de ces 2 options, malheureusement la gestion dynamique du nombre de requêtes envoyées ainsi que le ramp-up rendent cela plus difficile à exploiter avec des outils existants. Alors il est temps de sortir de se remonter les manches, il va falloir écrire du code !

curl

- Première version (0.1) en novembre 1996
- Intégré depuis PHP 4.0.2 (Août 2000)

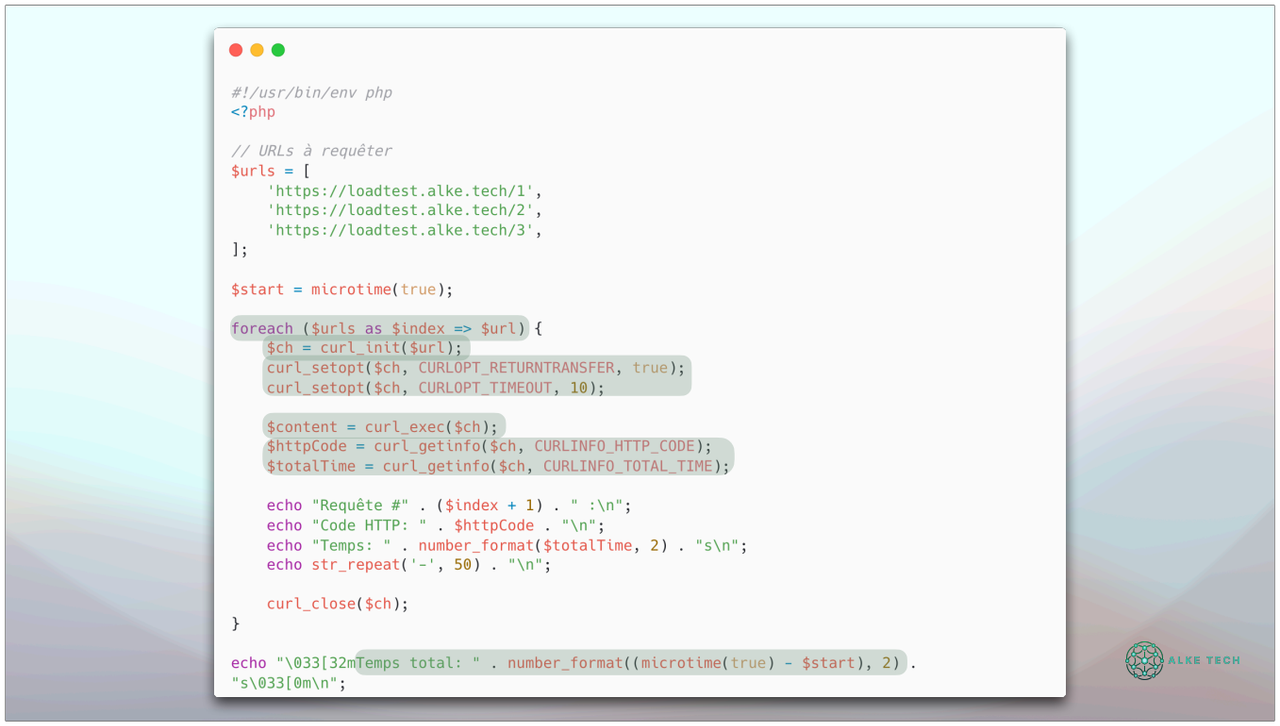
🎉 **Boring tech** 🎉

ALKE TECH

- La toute première version de curl a été publiée en version 0.1 en novembre 96
- C’est dispo avec PHP depuis 4.0.2 (aout 2000)
- Ça n’est pas la seule option permettant de faire des requêtes http, mais c’est la plus puissante: mise à jour depuis près de 30 ans, elle intègre au fur et à mesure les nouvelles technologies

Parmi les alternatives on peut notamment citer les streams en PHP, ainsi que file_get_contents par exemple.
Guzzle et HttpClient utilisent tous les 2 par défaut curl lorsque c’est disponible et peuvent faire du fallback lorsque ça n’est pas le cas.

Voyons maintenant comment curl s’utilise directement.



- examples/step1-curl

Une astuce très simple à avoir en tête lorsqu'on souhaite faire plusieurs requêtes successives vers le même serveur avec curl est de réutiliser la même ressource, afin de pouvoir réutiliser facilement la connexion.
Ici on a pratiquement le même script que précédemment, sauf qu'au lieu d'appeler curl_init à chaque itération, on va conserver notre ressource et venir modifier les options, comme par exemple l'url. Pour éviter d'avoir des options qu'on ne surcharge pas, curl_reset est bien pratique car **ça n'affecte ni la connexion ni la résolution dns**, mais par contre on va pouvoir repartir à 0 sur la requête.
Voici ce que ça donne comme durée totale.
C'est donc un changement mineur mais on voit qu'il a un impact assez important sur le temps total d'exécution de notre script. Cependant on continue d'effectuer nos requêtes successivement, or il est également possible avec curl de les exécuter en parallèle. Voyons comment.

```php
// URLs à requêter en parallèle
$urls = [...];

// Initialiser le gestionnaire multi
$multiHandle = curl_multi_init();
$curlHandles = [];

$start = microtime(true);

foreach ($urls as $index => $url) {
    $ch = curl_init($url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_TIMEOUT, 10);

    // Ajouter le handle au gestionnaire multi
    curl_multi_add_handle($multiHandle, $ch);
    $curlHandles[$index] = $ch;
}

// Exécuter toutes les requêtes en parallèle
$running = 0;
do {
    curl_multi_exec($multiHandle, $running);
    curl_multi_select($multiHandle);
} while ($running > 0);

// Récupérer les résultats
$results = [];
foreach ($curlHandles as $index => $ch) {
    $content = curl_multi_getcontent($ch);
    $httpCode = curl_getinfo($ch, CURLINFO_HTTP_CODE);
    $totalTime = curl_getinfo($ch, CURLINFO_TOTAL_TIME);

    echo "Requête #" . ($index + 1) . " :\n";
    // ...
    echo str_repeat('-', 50) . "\n";

    // Nettoyer
    curl_multi_remove_handle($multiHandle, $ch);
    curl_close($ch);
}

// Fermer le gestionnaire multi
curl_multi_close($multiHandle);

echo "\033[32mTemps total: " . number_format((microtime(true) - $start), 2) . "s\033[0m\n";
```


Bien sûr les exemples précédents ne fonctionnent correctement qu'en cas de requêtes qu'on souhaite faire de manière successive. Auquel cas on peut économiser l'établissement de la connexion TCP ainsi que la négociation TLS, mais on aura toujours la somme de chaque requête comme durée totale pour notre script.
Il est possible également d'effectuer plusieurs requêtes simultanément avec curl. Pour cela on initialise une ressource curl multiple et on va venir associer à cette ressource curl multiple les différentes requêtes unitaires qu'on souhaite réaliser.
- examples/step3-curl-multi.php

Le busy looping

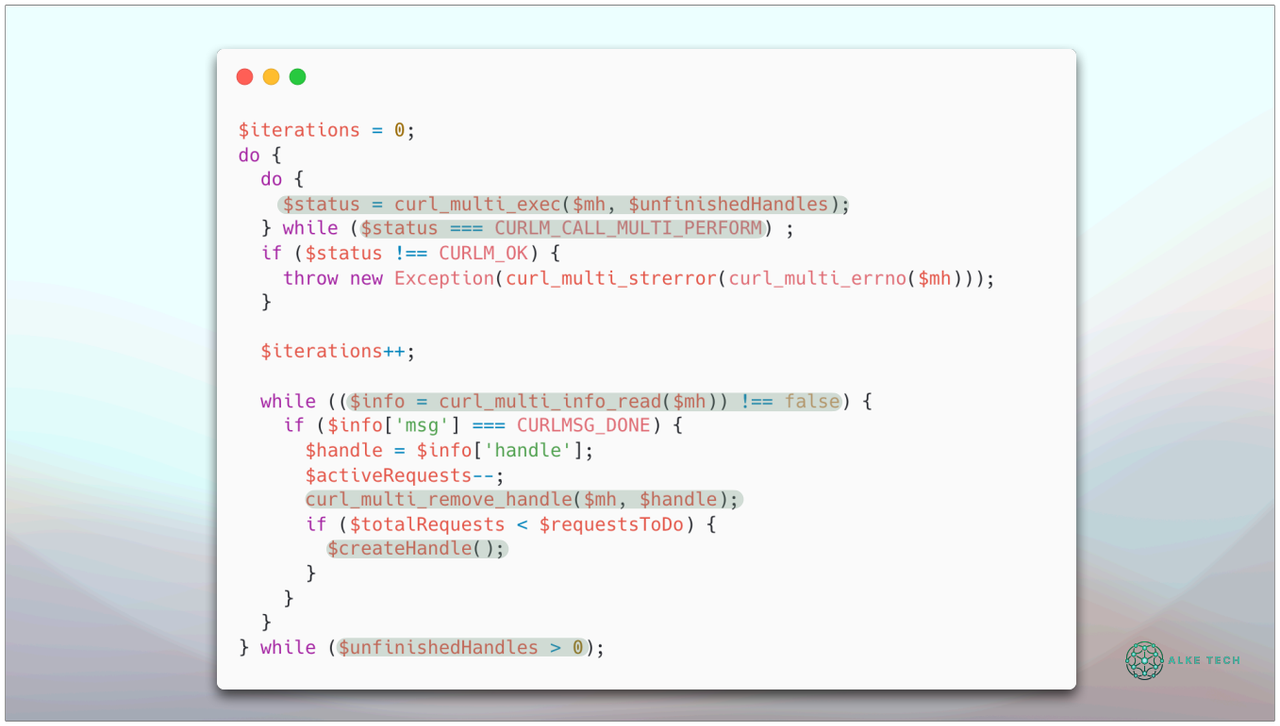
Une boucle coûte plus cher qu'il n'y parait

Un busy loop est une boucle qui consomme continuellement du temps processeur en vérifiant de manière répétée une condition, sans jamais se mettre en pause.

ALKE TECH

Il y a un gros point d'attention globalement quand on commence à travailler avec curl_multi et globalement avec ce que nous allons faire aujourd'hui: le busy looping.
Imaginez que vous attendez un sms important. Si vous regardez votre téléphone toutes les secondes pendant 1h pour voir s'il est arrivé, vous n'aurez globalement rien le temps de faire d'autre durant cette heure.
Alors que si vous attendez que votre téléphoner sonne pour le regarder, vous aurez pu faire autre chose pendant cette heure là. Votre sms ne sera pas arrivé plus tôt, mais vous aurez fini votre pull request.

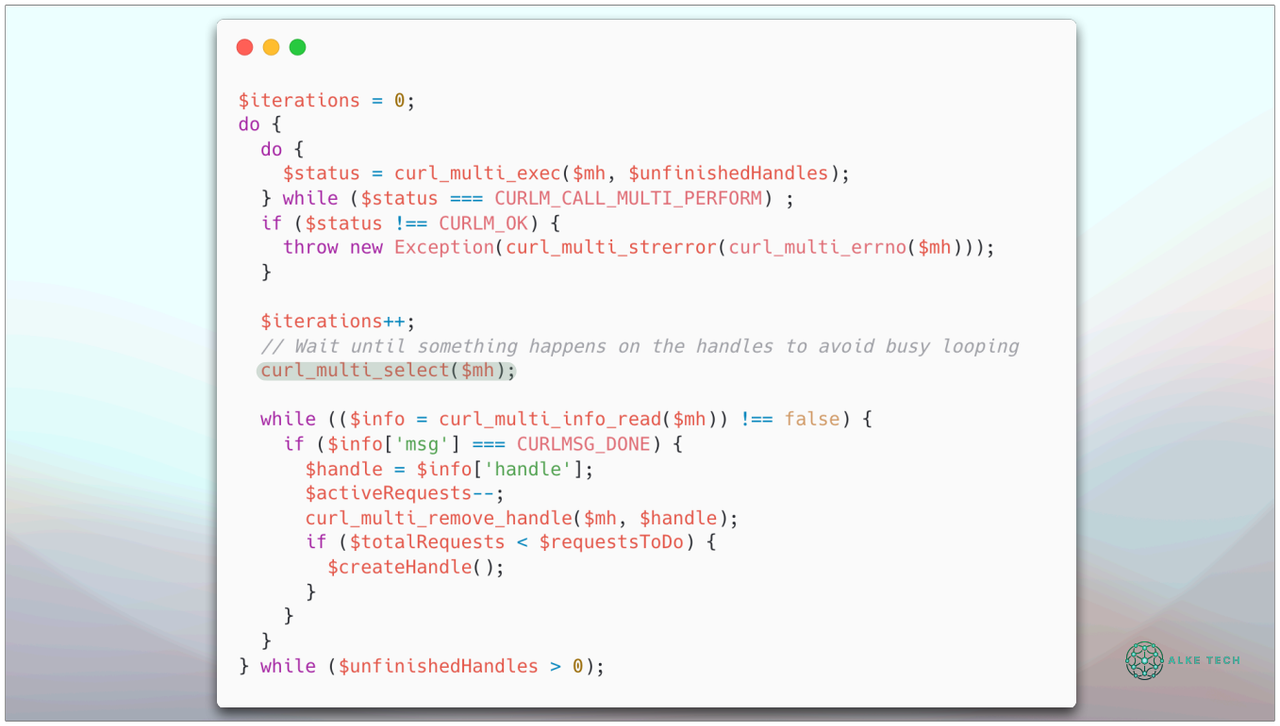
Heureusement avec curl_multi il y a des solutions très simples pour ne l'interroger que si quelque chose s'est passé.
Exemple concret ==> phpstorm
Bin/step3-busylooping.php et bin/step3-busylooping.php —avoid



L’implémentation précédente de notre boucle avait pour particularité d’attendre la fin de l’exécution de l’ensemble des requêtes avant de commencer à itérer sur les résultats, essentiellement pour pouvoir récupérer les réponses dans le même ordre que les requêtes, pour notre démo.
En pratique ce qu’on va plutôt faire c’est lire les réponses au fur et à mesure de leur disponibilité. Pour cela:

1. On continue d’appeler curl_multi_exec comme dans l’exemple précédent
2. Ensuite on va lire les réponses disponibles à l’aide de curl_multi_info_read, tant qu’il y en a
3. On retire le handler curl
4. Et on peut relancer une nouvelle requête immédiatement

Par contre notre algo à un soucis: on exécute en boucle les fonctions curl, même si rien ne s’est passé entre 2 interrogations.



Exemples/step4-busylooping.php et step4-busylooping.php —avoid

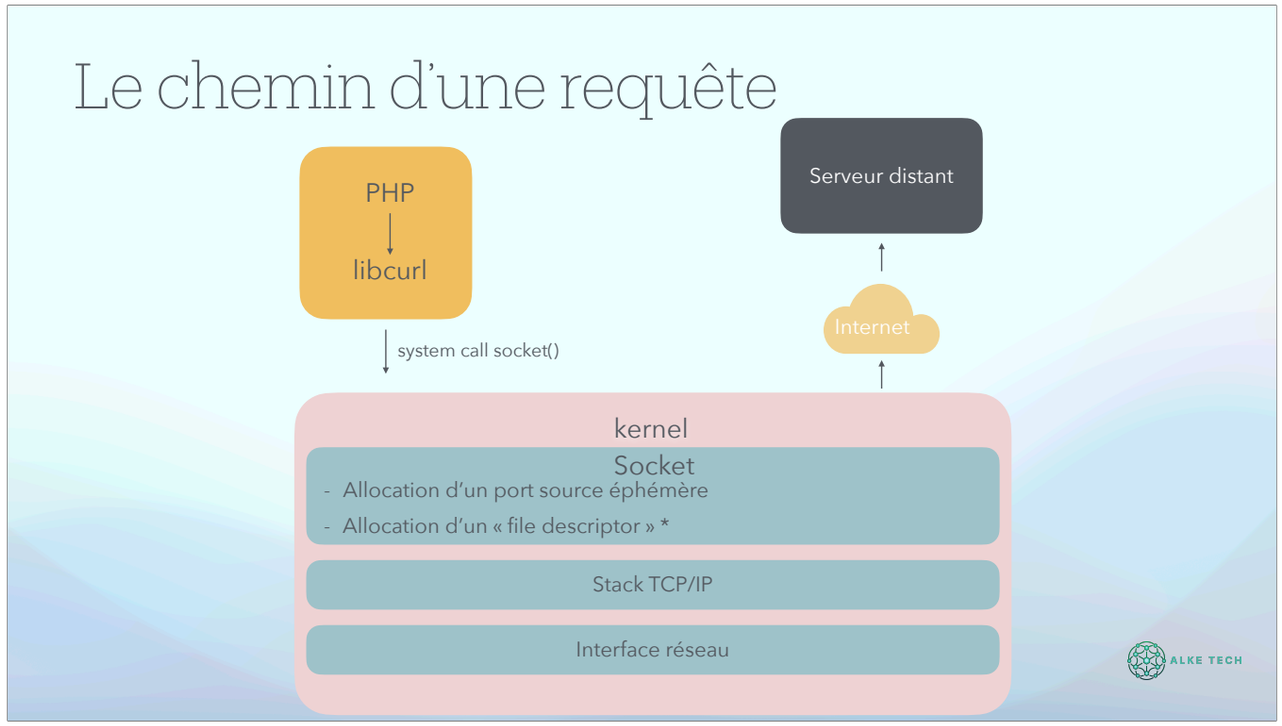
Maintenant que nous avons les bases de l’usage de curl, il temps d’aller un peu plus loin et commencer à faire un peu plus de requêtes. On se rappelle qu’on cherche à en faire 180 000 par seconde, il faut donc qu’on muscle un peu notre jeu.
Avant de commencer à aller très très fort dans les requêtes, il y a un point auquel il va falloir qu’on fasse particulièrement attention. J’en ai parlé rapidement un peu plus tôt, l’établissement d’une connexion ça prend un peu de temps.

Le mur de connexions

N'est jamais très loin

ALKE TECH

Chercher à ouvrir trop de connexions d’un coup risque de nous envoyer dans le mur, on va soit avoir du mal à obtenir des ressources côté client ou côté serveur avec un backlog qui va se remplir plus vite que le serveur web ne pourra le vider. Il faut donc prendre garde à ce point et établir progressivement les requêtes afin d’éviter des erreurs d’établissement des connexions. Pas de règle absolue ici, ça va dépendre de l’infrastructure en face et de ce qu’on cherche à atteindre.
Voyons comment ça se passe en détail

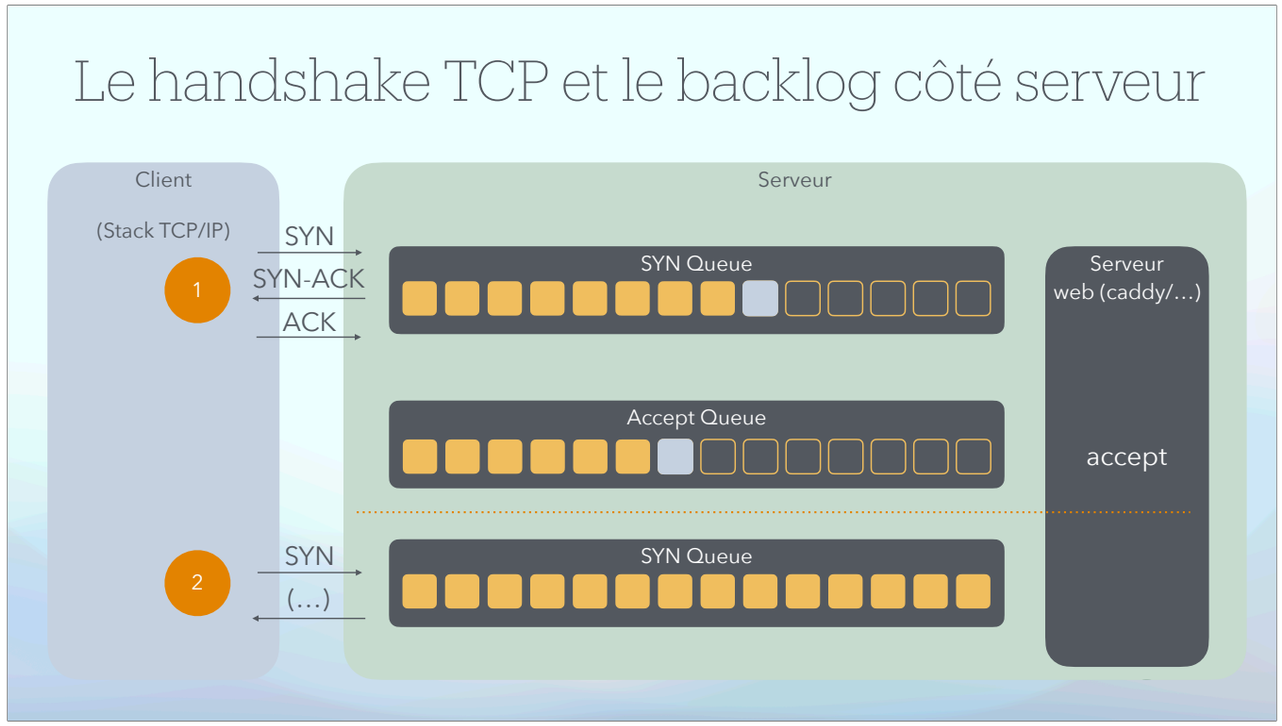


Lorsqu'on cherche à émettre une requête vers un serveur web distant en php avec curl, nous allons avoir notre script php qui, par le biais de l'extension standard, va appeler la libcurl, puis elle même va appeler la fonction socket exposée par le kernel. La socket va être créée par le kernel, qui va nous attribuer un port source éphémère ainsi qu'un file descriptor

* everything is a file

Ensuite le kernel va appeler la stack TCP/IP pour préparer l'établissement de la connexion, l'interface réseau va se charger d'envoyer tout ça via notre cable ethernet / fibre ou wifi, avant que ça ne sorte sur internet et que ça accède au serveur distant.

Si on zoome un peu sur la partie TCP/IP, voici ce qui va se passer.



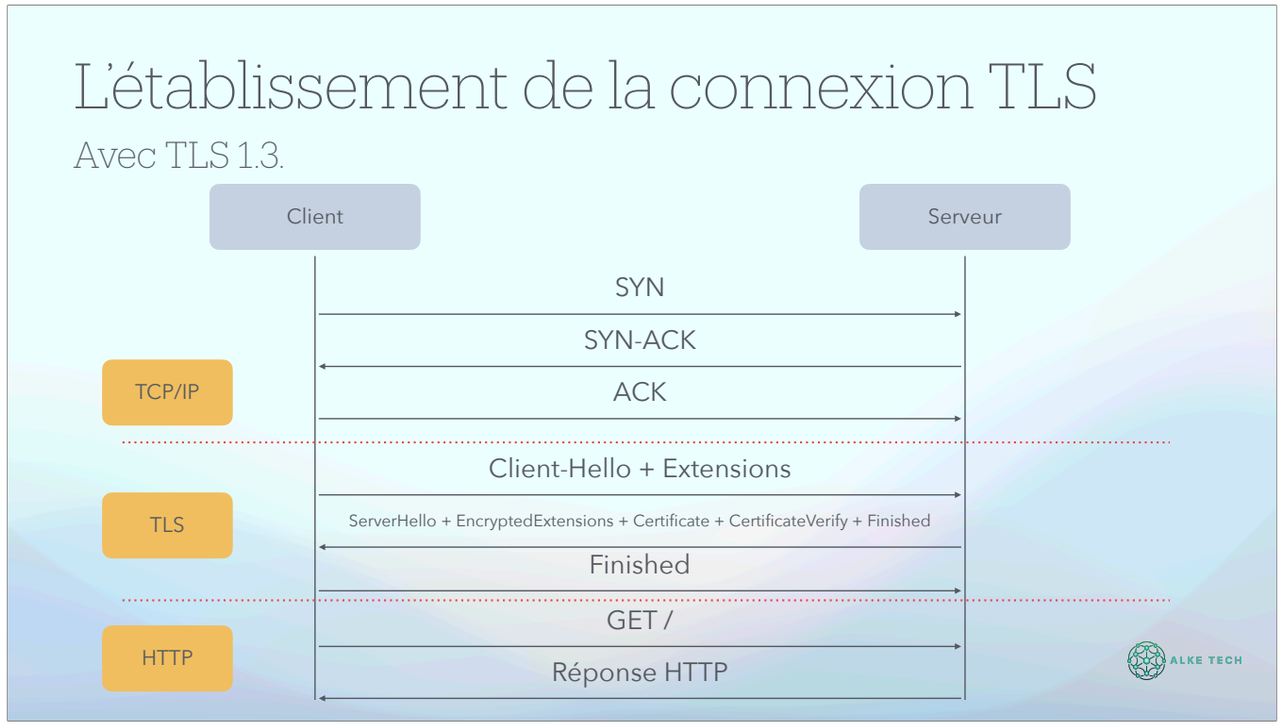
Si on regarde ce qui se passe au niveau de la stack TCP/IP pour établir la connexion en détails, voici ce que ça donne.

Tout d'abord le client envoie un paquet « SYN », il s'agit d'une demande de connexion.
Si il y a encore de la place, elle s'inscrit dans la Syn Queue.
Le serveur répond alors avec un paquet SYN-ACK. Le client répond enfin avec un paquet ACK. La connexion est alors établie, on considère que les paquets arrivent à circuler dans les 2 sens.
Imaginons 2 personnes qui souhaitent se parler mais sont un peu loin: « Alice, je veux te parler, tu m'entends ? » - « Oui Bob, et toi ? tu m'entends ? » - « Oui Alice je t'entends c'est bon. »
La demande de connexion va alors dans l'Accept Queue jusqu'à ce que l'application qui écoute accepte la connexion.

Ensemble, la SYN Queue et l'ACCEPT QUEUE représentent le backlog, dont la taille est limitée par un paramètre kernel.
Si la SYN QUEUE est pleine, le serveur ne répondra jamais avec le SYN-ACK.
Si l'accept queue est pleine, alors la connexion sera refermée sans être transmise au serveur web.

```php
$stepsForConnections = 50;
$iterationsOfNewConnections = 0;
do {
    if ($this->workerStats->activeRequests < $this->maxParallelRequests && !$this->isPaused()) {
        $requestsToCreate = min((++$iterationsOfNewConnections) * $stepsForConnections, $this->maxParallelRequests - $this->workerStats->activeRequests);
        for ($i = 0; $i < $requestsToCreate; $i++) {
            if (!$createHandle()) {
                break;
            }
        }
    }
    do {
        $status = curl_multi_exec($this->mh, $unfinishedHandles);
    } while ($status === CURLM_CALL_MULTI_PERFORM);
    if ($status !== CURLM_OK) {
        throw new Exception(curl_multi_strerror(curl_multi_errno($this->mh)));
    }
    while (($info = curl_multi_info_read($this->mh)) !== false) {
        // Traiter les réponses
    }
} while ($this->workerStats->activeRequests > 0 || $hasMoreMessage());
```


Avec cet algo on va établir progressivement de nouvelles connexions, l'idée est donc que petit à petit on va s'autoriser à envoyer de plus en plus de nouvelles requêtes simultanées, notamment car avec HTTP/2 et /3 on va pouvoir faire transiter beaucoup de requêtes avec une seule connexion.



Mais pour faire du http aujourd'hui il est assez rare que les échanges d'informations ne soient pas chiffrées, il faut donc également mettre en place ce chiffrement avec à nouveau des échanges d'informatons.

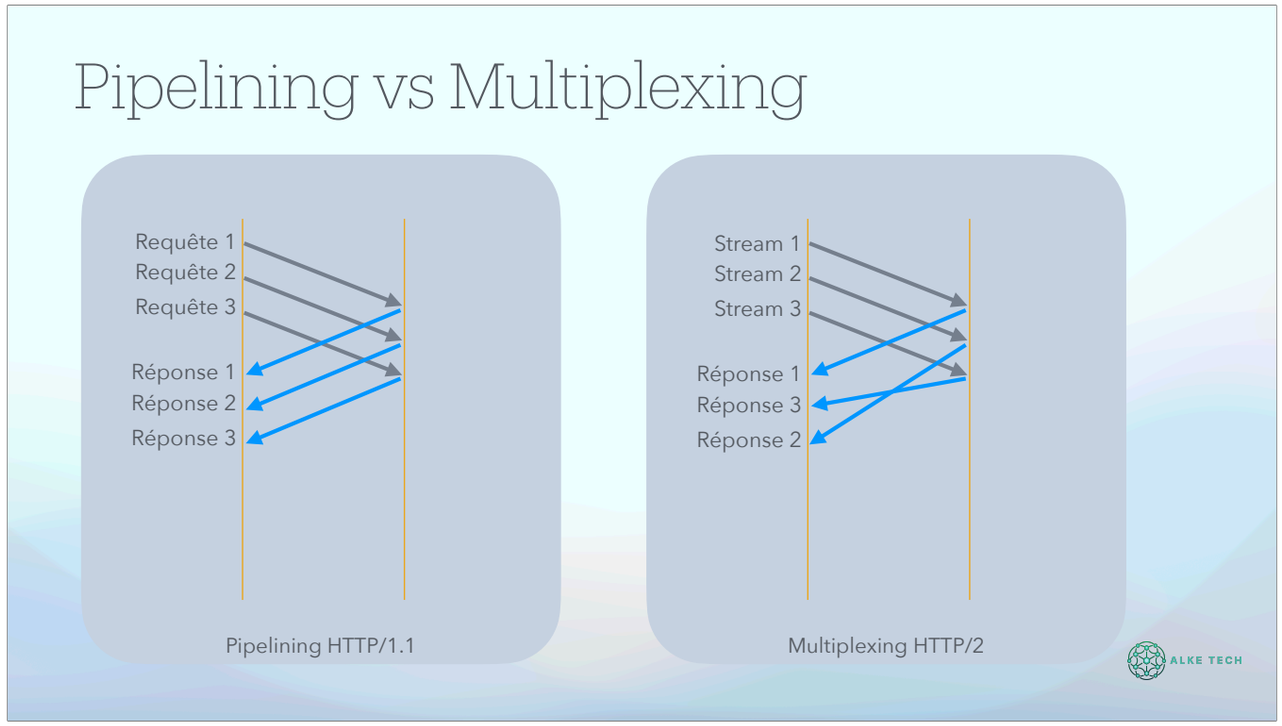
Avec TLS 1.3, voici ce que ça donne, on a toujours l'établissement de notre connexion TCP au départ avec:
1. Le client envoie son paquet SYN
2. Le serveur répond SYN-ACK
3. Le client répond ACK

Puis lorsque la connexion est établie, le serveur web s'occupe de la négociation TLS avec le client: le client indique ce qu'il supporte comme protocoles et le serveur répond avec le certificat et les clés permettant de continuer l'échange de manière chiffrée. C'est seulement après que notre requête HTTP réelle va transiter. Par exemple un GET / pour faire une requête GET vers la homepage.

HTTP et les connexions

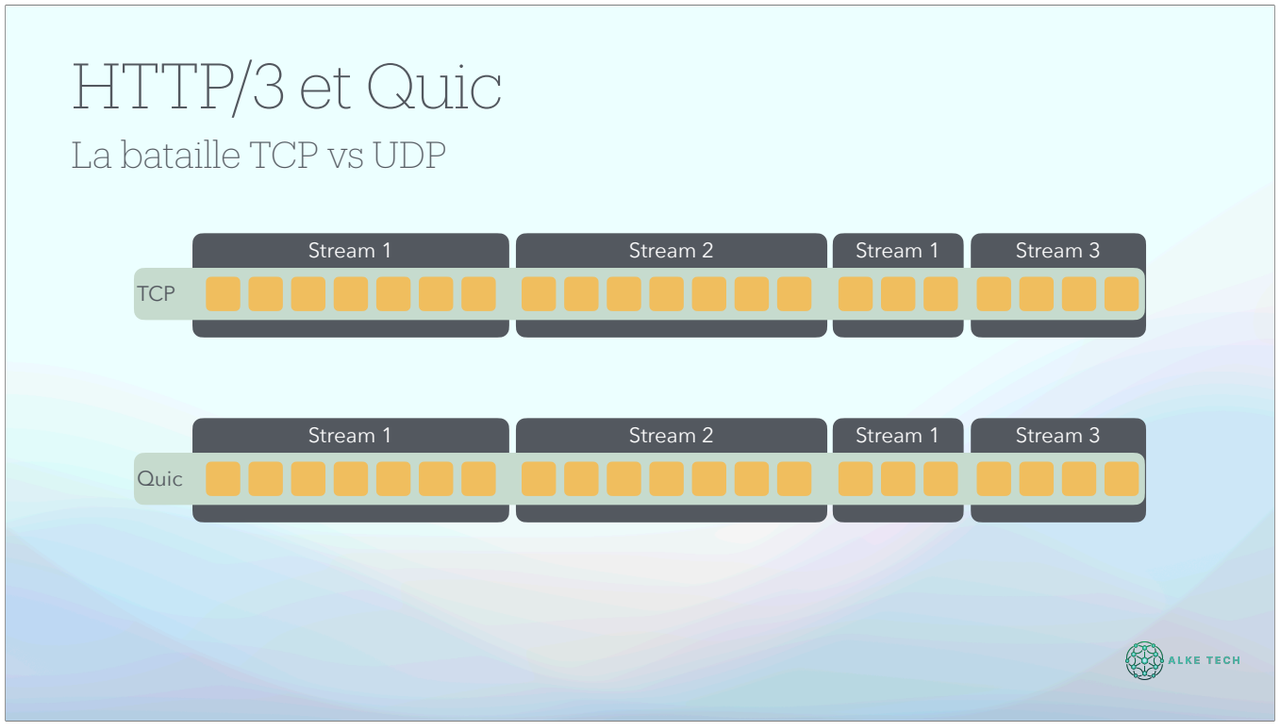
ALKE TECH

Et la parallélisation au niveau HTTP, comment ça se passe dans tout ça ?

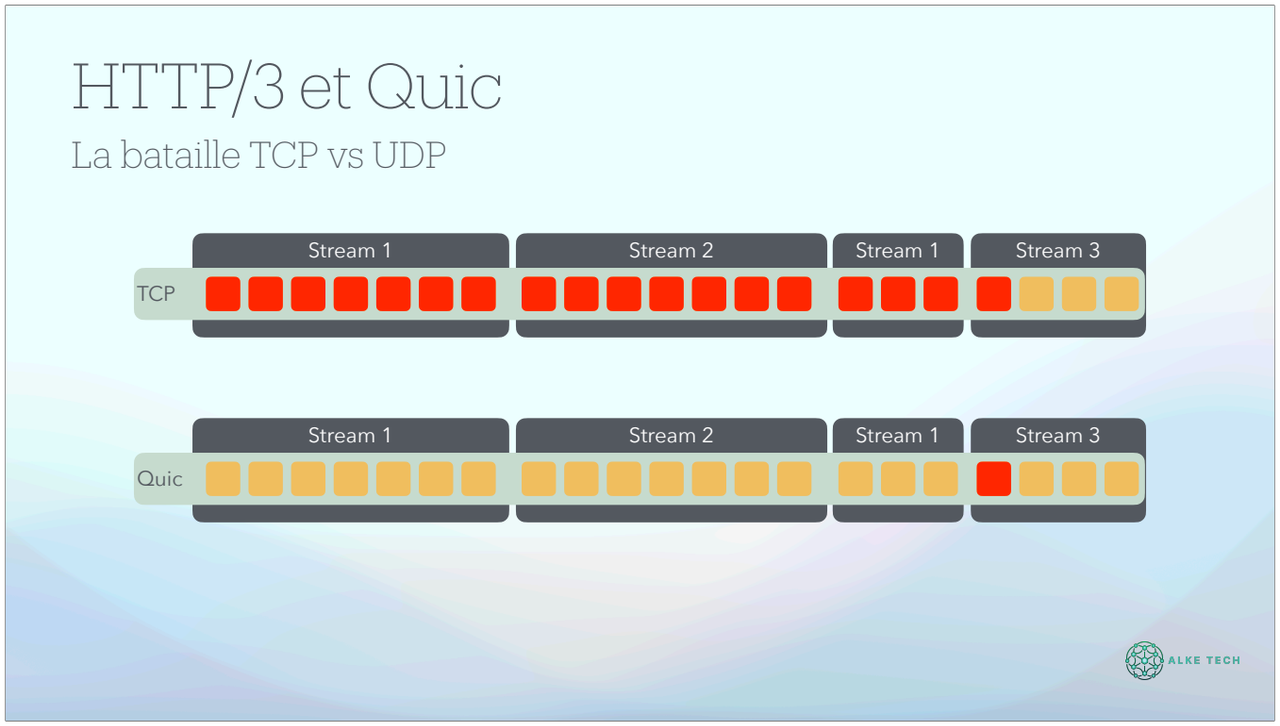


Le pipelining était utilisé avec HTTP/1.1, il permettait d'envoyer successivement plusieurs requêtes puis de recevoir toutes les réponses dans le même ordre. Cela était une bonne idée mais en pratique ça fonctionne assez mal, notamment en raison du côté séquentiel et donc bloquant. Plein de reverses proxy ne supportent pas réellement le pipelining et en cas d'échec d'une requête, la gestion d'erreur est complexe. On s'appuie donc plutôt sur le keepalive pour faire plusieurs requêtes sur une seule connexion avec HTTP/1: lorsque la requête précédente est terminée on envoie la nouvelle. Comme c'est tombé un peu à l'eau, notamment avec les navigateurs qui ne l'utilisaient pas, on est passé à autre chose.
HTTP/2 a apporté le multiplexing avec une notion de streams: de nombreux streams sont exploités au sein d'une seule connexion, il s'agit d'autant de flux dans lesquels on peut faire transiter des requêtes et des réponses indépendamment les unes des autres.
Ce mécanisme très efficace permet de réduire le besoin de connexions avec H/2 puisque de nombreuses requêtes vont transiter.
C'est réellement à partir de HTTP/2 qu'on a une décorélation entre le nombre de requêtes simultanées et le nombre de connexions ouvertes.
Il y a toujours un problème néanmoins lié au TCP et la gestion de la perte des paquets.

Il a souvent été dit que HTTP/2 n’était pas utile dans des connexions entre serveurs, c’est faux notamment dans le cadre de requêtes simultanées, optimisant la quantité d’informations transmises et le nombre de connexions.



Regardons ce flux et imaginons ces paquets qui arrivent au sein d’une connexion HTTP/2. La connexion utilise donc le protocole TCP. La couche HTTP va gérer de récupérer les morceaux et les assembler en réponses cohérentes grâce au stream id. Au niveau TCP on va avoir notamment la gestion de la retransmission des paquets. Aucune notion des streams, c’est géré à un niveau au dessus.
TCP gère les paquets de manière ordonnée. Cela signifie que lorsqu’un paquet est corrompu et doit être retransmis, tous les paquets transmis après doivent être retransmis.



Donc cela signifie que si le dernier paquet du stream #3 est corrompu, tous les paquets des streams 1 et stream 2 doivent être retransmis. Alors même que les réponses pouvaient être complètes à ce stade.
Ça c'est donc lié au fait que le protocole qui établit la connexion nous assure de ne transmettre à l'application que des données correctes et vérifiées. Ce faisant il n'a pas connaissance du détail de ce que l'application fait transiter dans ces paquets: ici des requêtes et des réponses regroupées sous forme de streams. On appelle ce problème le blocage HOL: Head Of Line Blocking

UDP est un autre protocole de communication, une alternative avec TCP, qui a 2 différences fondamentales:
- Établissement de connexions plus léger: pas de handshake contrairement à ce qu'on a vu précédemment
- Pas de retransmission de paquets corrompus.

Cela signifie que l'application va donc recevoir des données non fiables, il faut que cet aspect soit géré différemment. Et c'est là que Quic entre en jeu, puisque Quic possède à la fois la connaissance concernant les paquets, mais également des informations de lus haut niveau comme le stream concerné par ce paquet. Et ce protocole mets en place une retransmission sur la base du paquet corrompu et uniquement celui-ci.
Par ailleurs on peut noter dans Quic que globalement il induit des tailles de paquet plus petite, ce qui signifie d'avantage de paquets à faire transiter et d'avantage de consommation CPU afin de vérifier la somme de contrôle de ces paquets.

Les différences entre Quic et TCP

TCP

- Géré entièrement par le kernel
- Pas de support de TLS (géré plus haut)
- HOL Blocking
- Faible consommation CPU et réseau
- 1 aller/retour pour connexion en clair (+ 1 pour le TLS par l'application)
- Connexion coupée en cas de changement d'IP

Quic

- S'appuie sur UDP (kernel) + userland
- Support natif de TLS
- Retransmission unitaire de paquets
- Surconsommation CPU et réseau
- 1 aller/retour pour connexion chiffrée
- Support changement d'IP (Wifi <-> 5G)

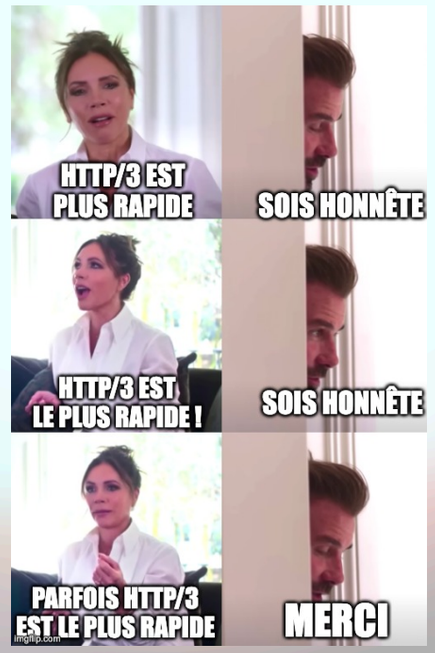
ALKE TECH

HTTP: le match

	HTTP/1.1	HTTP/2	HTTP/3
Protocole réseau	TCP / IP	TCP / IP	Quic (UDP / IP)
Sécurité	Optionnelle	Obligatoire*	Obligatoire et native
Parallélisation	Difficile	Bonne	Très bonne
Format d'échange	Ascii	Binaire	Binaire
Compression des headers	Aucune	Bonne HPack	Très bonne QPack
Fonctions avancées		Server-Push / Early-Hints Priorisation	Server-Push** / Early-Hints Priorisation

*: De facto obligatoire car pas dans le standard, mais côté navigateurs H/2 n'est implémenté qu'avec du TLS. Côté client le support sans TLS varie et fait souvent l'objet d'options spécifiques pour autoriser le « h2c »: h2 cleartext

** Les serveurs push sont conditionnels avec HTTP/3, le client doit avoir déclaré qu'il était d'accord pour recevoir des push + un nombre maximum que le serveur devra respecter.


HTTP/3 EST
PLUS RAPIDE
SOIS HONNÊTE
HTTP/3 EST
LE PLUS RAPIDE !
SOIS HONNÊTE
PARFOIS HTTP/3
EST LE PLUS RAPIDE
MERCI
imgflip.com

ALKE TECH

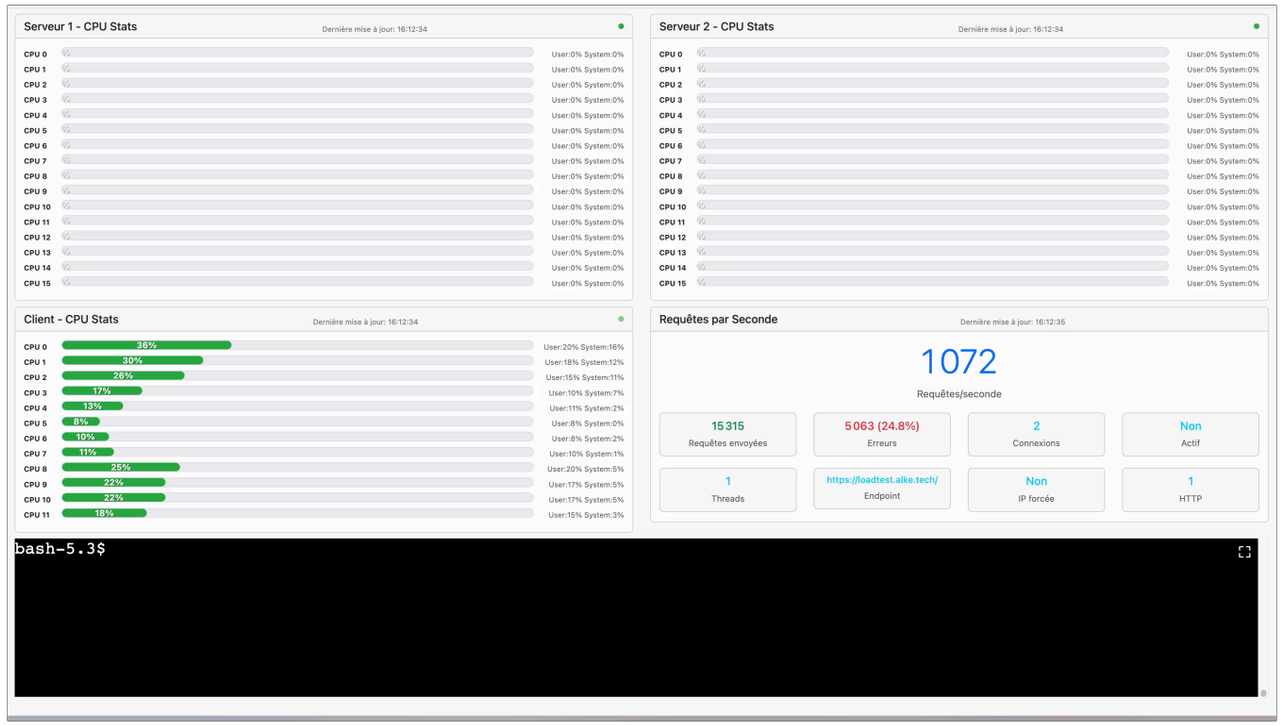
Reality check: **parfois** HTTP/2 reste plus rapide

Le support de HTTP/3 dans les serveurs est désormais assez bon, les principaux serveurs web le supportent (nginx et caddy en officiel, apache en expérimental). Les browser le supportent bien, après ça peut être plus périlleux. Je n'ai pas trouvé d'outil récent qui permettent de faire des loadtests http/3 vs http/2.
Côté PHP j'ai identifié un bug soit dans libcurl, soit dans l'extension curl pour php qui empêche la réutilisation de handlers et donc de faire fonctionner à fond notre code. Le workaround coûte + cher en CPU.

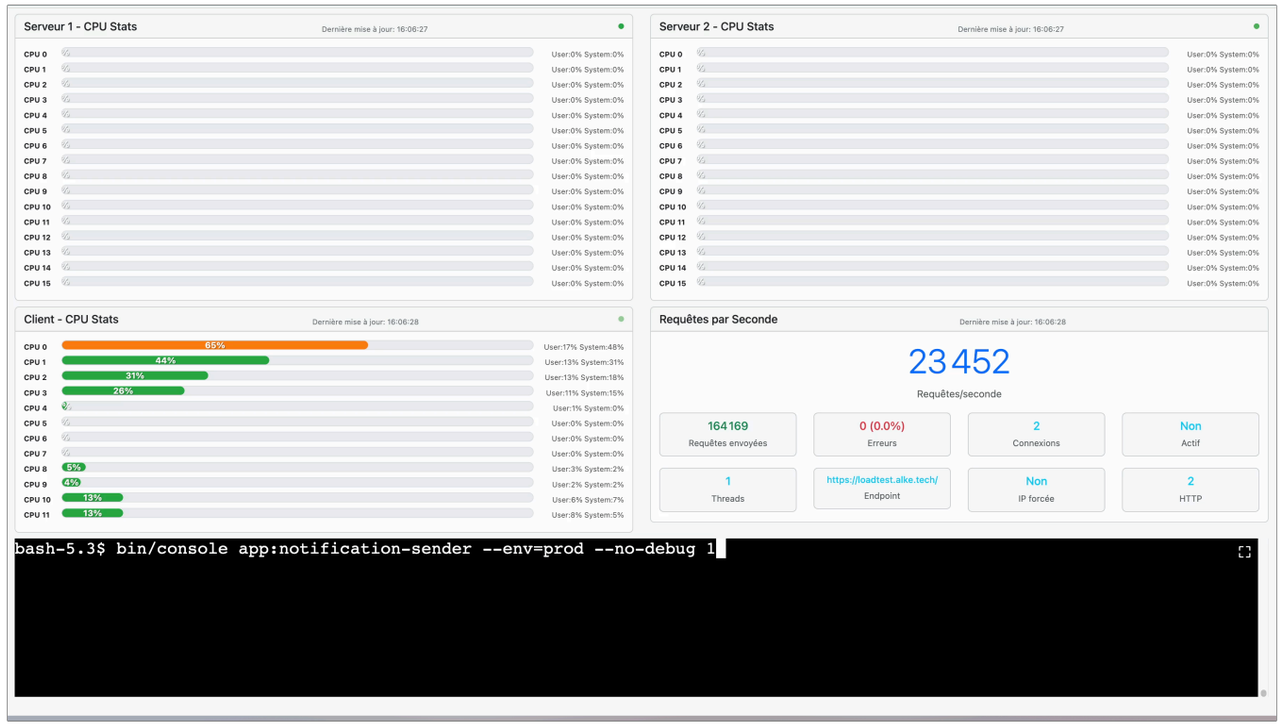
Cependant pour de la communication serveur à serveur avec une très forte parallélisation: HTTP/2 sera probablement meilleur:
- Moins d'overhead lié au protocole pour la retransmission des paquets, gain sur connexion stable malgré blocage HOL
- Moins de CPU lié aux contrôle des paquets: si on est limité par le CPU (comme dans notre test actuel): gain vs H3

L'implémentation de HTTP/3 dans curl est pour le moment assez fragile, différentes librairies peuvent être utilisées par curl pour gérer H/3 et ça n'est pas encore 100% mature. Cela n'est pas très étonnant puisque curl étant massivement utilisé pour faire du serveur à serveur, ça n'est pas au sein de cet outil que les gains seront les plus importants.

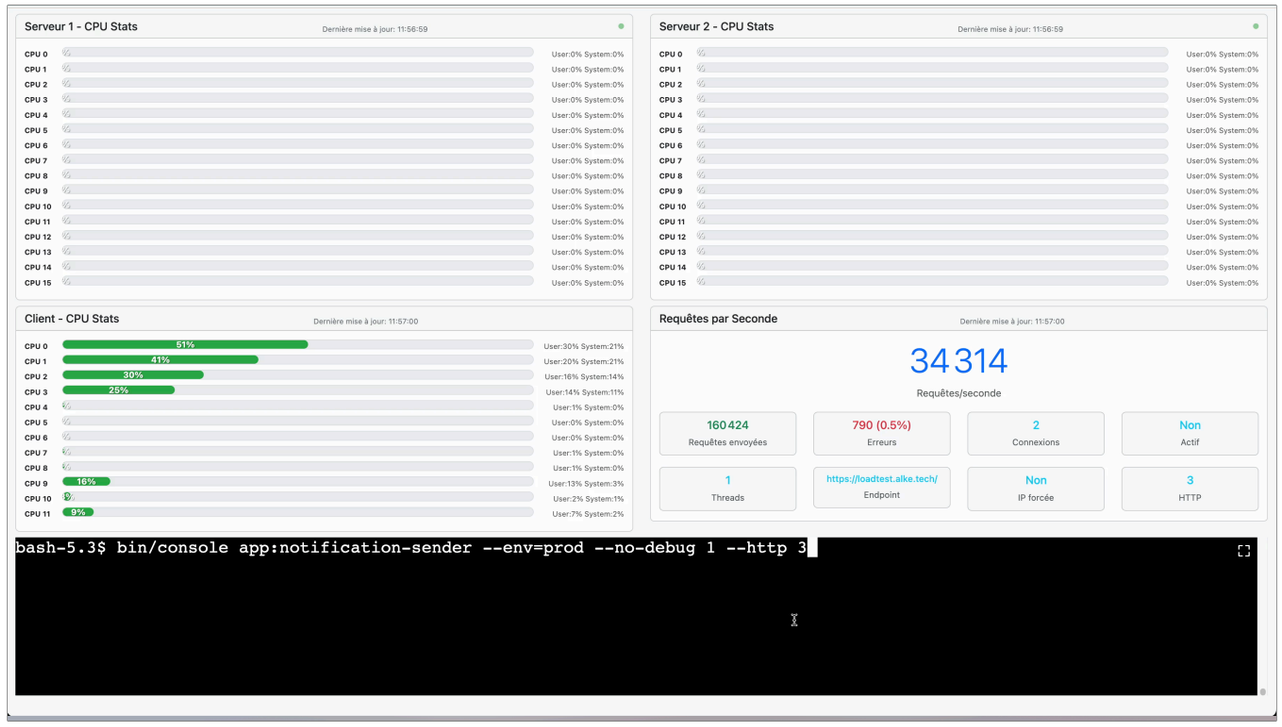
bin/console app:notification-sender 1 --http=1 --env=prod —no-debug
bin/console app:notification-sender 1 --http=2 --env=prod --no-debug
bin/console app:notification-sender 1 —http=3 --env=prod --no-debug



THREAD: 1
HTTP: 1
PAS DE FORCAGE DNS



THREAD: 1
HTTP: 2
PAS DE FORCAGE DNS



THREAD: 1
HTTP: 3
PAS DE FORCAGE DNS

Le DNS est le système d’annuaire d’internet. C’est grâce à ce protocole et l’infrastructure associée qu’on peut connaître la ou les adresses IP associées à un nom d’hôte.
Dans notre exemple on utilise loadtest.alke.tech.
Vérifions les enregistrements DNS pour ce serveur:
> dig loadtest.alke.tech

Parfait, on voit 2 enregistrements A, ce qui indique que les 2 IP indiquées peuvent être jointes pour ce nom de domaine.

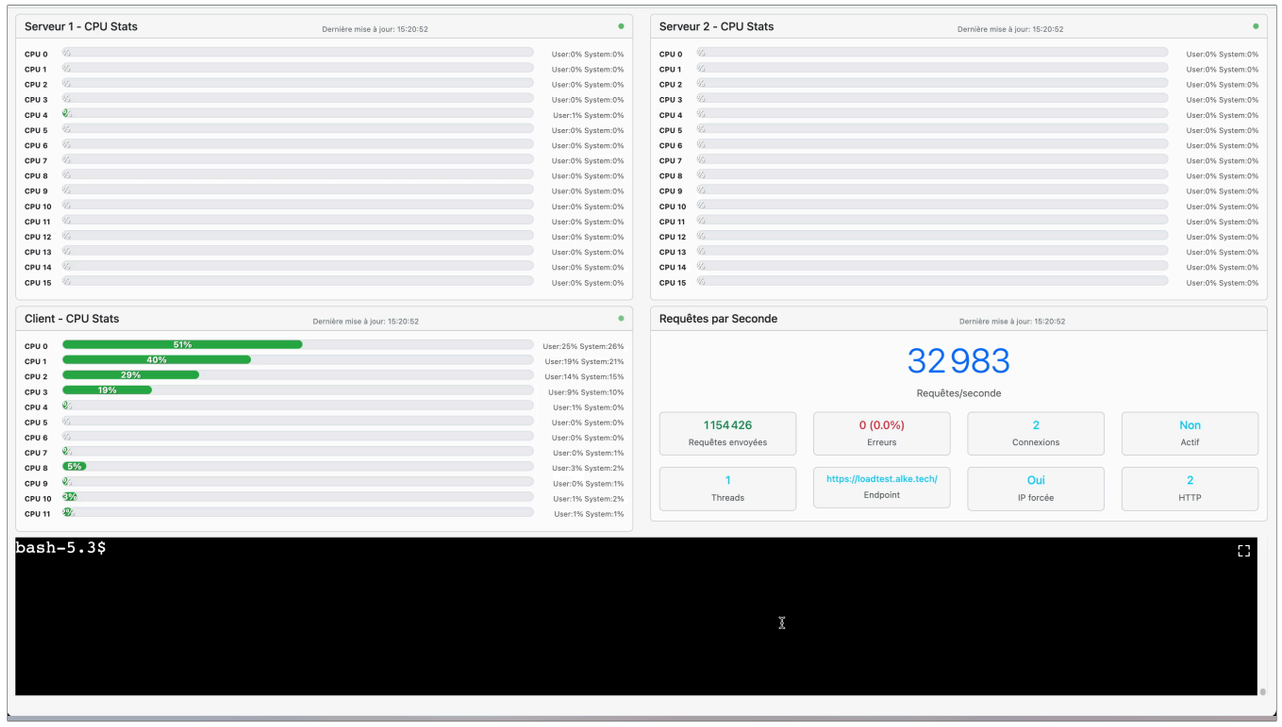
Ce qui est étonnant cependant, c’est que nous avons plusieurs IP mais une seule semble être utilisée par curl pour ses connexions. Comment faire dès lors pour répartir la charge entre les différents serveurs ? Quand on cherche à tabasser autant de requêtes, mieux vaut étaler.


private function resolveHostname(string $hostname): array|false
{
$ips = [];
try {
// Lookup address info for HTTP service with SOCK_STREAM hint
$addrInfos = socket_addrinfo_lookup($hostname, 'http', [
'ai_socktype' => SOCK_STREAM,
]);
if ($addrInfos === false) {
return false;
}
foreach ($addrInfos as $addrInfo) {
$explained = socket_addrinfo_explain($addrInfo);
if ($explained && isset($explained['ai_addr']['sin_addr'])) {
// IPv4 address
$ips[] = $explained['ai_addr']['sin_addr'];
} elseif ($explained && isset($explained['ai_addr']['sin6_addr']))
{
// IPv6 address
$ips[] = $explained['ai_addr']['sin6_addr'];
}
}
return empty($ips) ? false : array_values(array_unique($ips));
} catch (\Exception) {
return false;
}
}
ALKE TECH

```php
public function getIpForHostname(string $hostname): string
{
    if (filter_var($hostname, FILTER_VALIDATE_IP)) {
        return $hostname;
    }

    $hostname .= '.'; // Add a final dot to the hostname, to make it absolute
    if (!isset($this->ipsPerHostname[$hostname]) || $this->ipsPerHostname[$hostname] === false)
{        $this->ipsPerHostname[$hostname] = $this->resolveHostname($hostname);
        $this->indexToUsePerHostname[$hostname] = 0;
    }

    if ($this->ipsPerHostname[$hostname] === false) {
         throw new RuntimeException("Could not resolve hostname $hostname");
    }

    $this->indexToUsePerHostname[$hostname]++;
    if ($this->indexToUsePerHostname[$hostname] >= count($this->ipsPerHostname[$hostname])) {
        $this->indexToUsePerHostname[$hostname] = 0;
    }

    return $this->ipsPerHostname[$hostname][$this->indexToUsePerHostname[$hostname]];
}
```

On met en oeuvre ça avec curl, pour cela on va d'abord faire la résolution DNS, puis construire la chaine au format que curl attends: Nom de domaine « : » port de destination, ici 443 pour https, « : » ip à utiliser.
On transmet ensuite cette information à curl via l'option CURLOPT_RESOLVE

Regardons désormais ce que ça donne en terme de performance.
`bin/console app:notification-sender 1 --force-hostname-resolution --env=prod --no-debug`



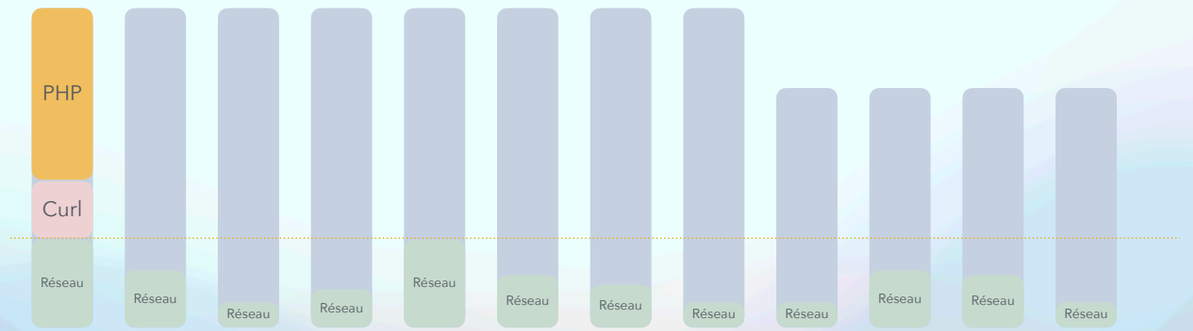
THREAD: 1
HTTP: 2
DNS: FORCÉ

Passage à l'échelle

Ou « Scaling » comme on dit d'habitude

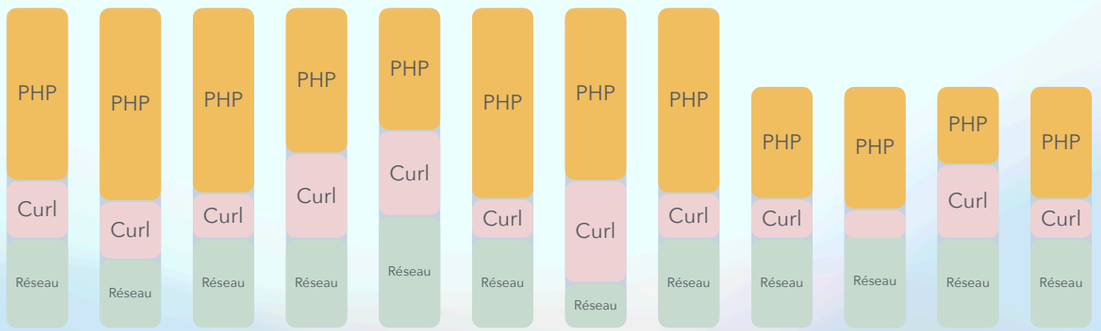
ALKE TECH

Usage du CPU: Point de départ



ALKE TECH

Usage du CPU: Objectif



ALKE TECH

Multi-threading et PHP

Les solutions possibles

- Pecl/pthreads
- Pecl/parrallel
- Ext/pcntl

ALKE TECH

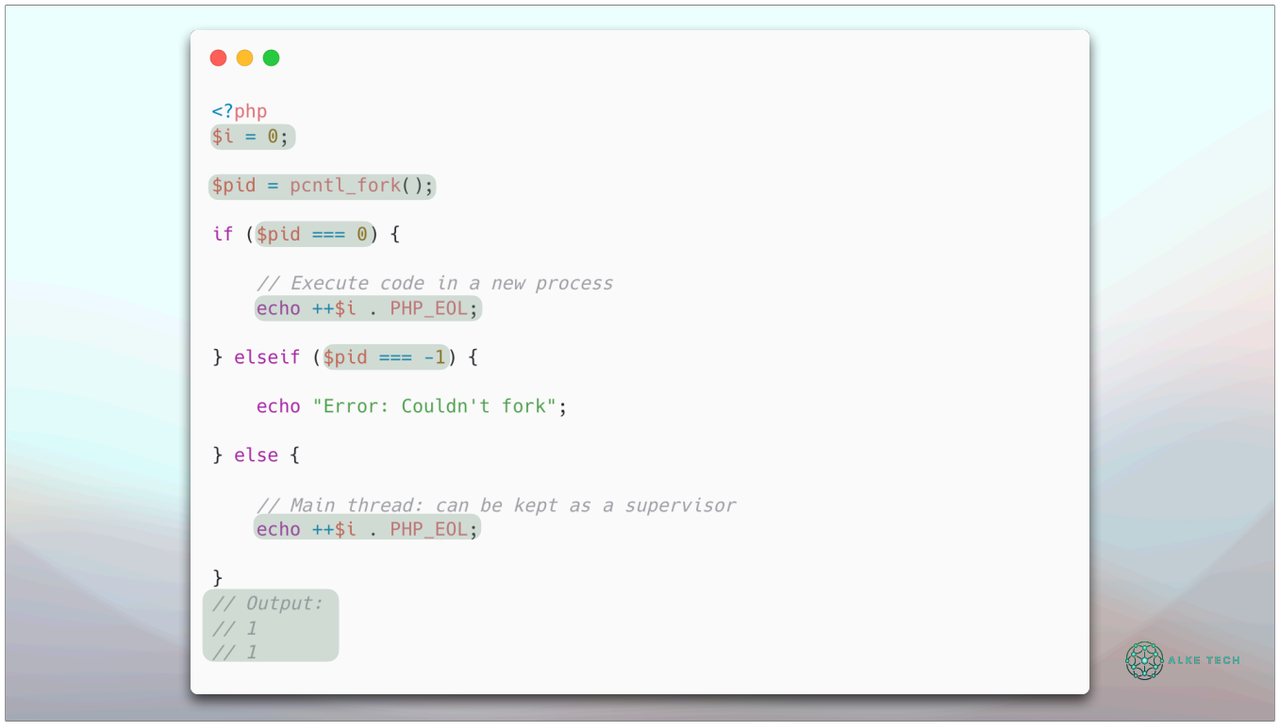
Multi-threading et PHP

Les solutions possibles

- ~~Pecl/pthreads~~ - abandonné avec PHP8
- Pecl/parrallel - extension supplémentaire, cross plateforme
- Ext/pcntl - extension native, pas disponible sur windows

ALKE TECH

Plusieurs solutions existent, notamment parrallel et pcntl. Parrallel est une bonne solution, elle ne sort pas de nulle part puisqu'elle est écrite par Joe Watkins, qui est contributeur de PHP et a été release manager ; on a donc affaire à quelque chose de solide, cependan parrallel va surtout nous permettre de venir écrire du code concurrent mais avec beaucoup de partages de structures internes au niveau PHP, ce qui n'est pas optimal ; alors que pctnl, encore une fois c'est de la boring tech puisque ça existe depuis toujours ; va nous permettre d'avoir une isolation très forte et une vraie parallélisation. Regardons son usage.




```php
<?php
$i = 0;

$pid = pcntl_fork();

if ($pid === 0) {

    // Execute code in a new process
    echo ++$i . PHP_EOL;

} elseif ($pid === -1) {

    echo "Error: Couldn't fork";

} else {

    // Main thread: can be kept as a supervisor
    echo ++$i . PHP_EOL;

}
// Output:
// 1
// 1
```

Le principe d’un fork en PHP arrive directement de C, où 2 chemins d’exécution de code vont divergé.
Le principe est le même qu’avec un repo git qu’on fork: on récupère une copie de tout, mais à partir de ce moment là on ne partage plus rien.
Voici un usage extrêmement basique de fork:

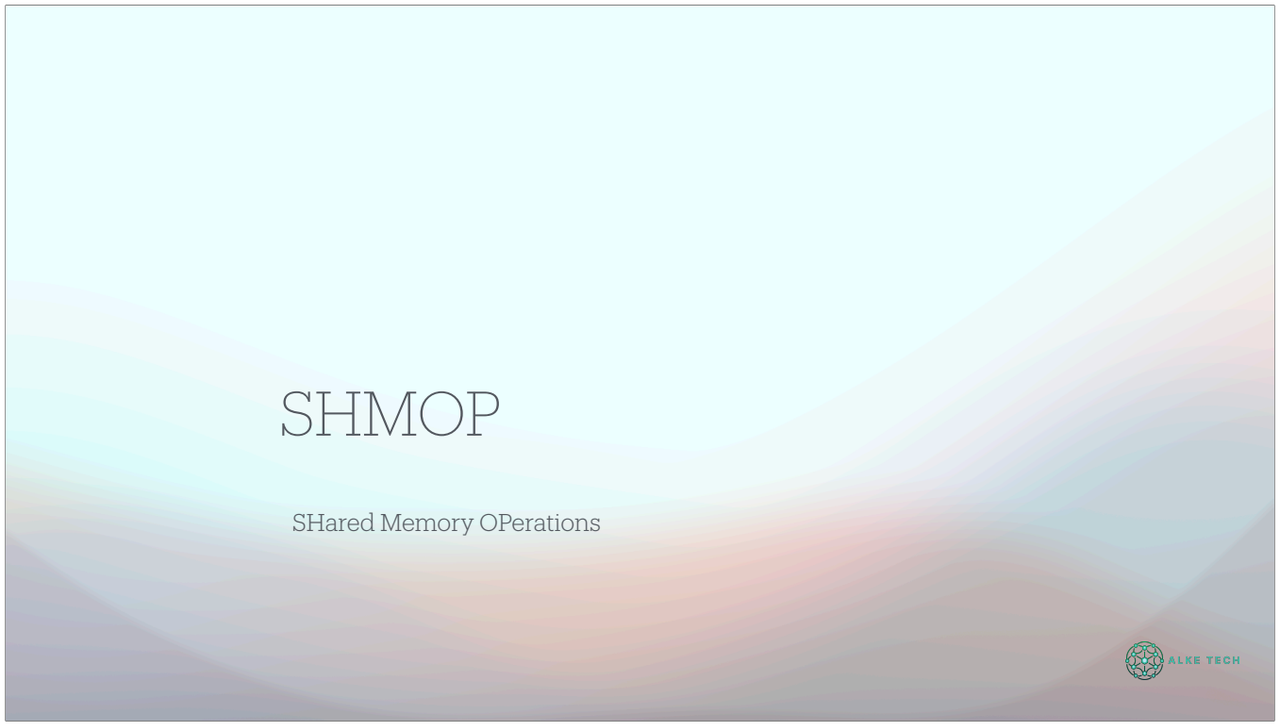
1. On initialise une variable à 0
2. On crée un fork, à partir de ce moment là processus différents s’exécutent, le second étant l’enfant du premier.
3. En fonction de la valeur de retour de la fonction fork, on va pouvoir savoir si on est au niveau dur parent ou si on est dans l’enfant et faire des choses différentes.
4.

```php
$i = 0;

$pid = pcntl_fork();

if ($pid === 0) {

    // Execute code in a new process

} elseif ($pid == -1) {

    echo "Could not fork" . PHP_EOL;

} else {

    // Main thread, can use as a supervisor
    $pid = pcntl_wait($status, WNOHANG);
    if ($pid > 0) {
        // Fork exited
        if (pcntl_wexitstatus($status) === 0) {
            // Ok, no problem here
        } else {
            // Oh, exit code means there is an error somewhere. Maybe
restart process ?
        }
    }
}
```

Par exemple côté parent on peut se mettre en superviseur / coordinateur et regarder le statut des enfants, pour attendre que tous les traitements soient terminés.
La fonction standard pcntl_wait, ici avec le paramètre WNOHANG pour dire qu'on souhaite un retour immédiat, va nous permettre de connaitre l'éventuel code de sortie du process. Si rien n'est retourné c'est que le process n'est pas sorti, on peut alors réinterroger la fonction plus tard. On peut également appeler la fonction sans le paramètre WNOHANG pour que la fonction soit bloquante.

Exemple: php exemples/step5-fork.php

L'approche que nous allons utiliser pour nos requêtes va donc être de créer un fork par coeur de CPU, les forks seront chargés d'effectuer les requêtes et le thread principal va rester en coordinateur pour vérifier l'état des forks, aggréger les stats, etc. Mais d'ailleurs les stats, comment on va les avoir ?



Notre prochain challenge une fois que nous avons plusieurs processus qui exécutent notre code, va être de pouvoir les faire discuter tous ensemble. Pour cela nous avons notamment la possibilité de créer des emplacements de mémoires partagée, dans lesquels nos process vont pouvoir lire et écrire.
L'extension SHMOP: Shared Memory Operations est là pour ça, elle s'appuie encore une fois sur ce qui existe plus bas niveau dans le système pour proposer ce mécanisme.
L'API proposée par cette extension standard est très simple, elle consiste en quelques fonctions permettant de créer un emplacement, lire écrire dedans ou le supprimer. Du CRUD pour la mémoire. Et comme c'est natif et qu'on travaille en mémoire, c'est super rapide.

Voyons ensemble à quoi ça ressemble.

```php
function executeInFork(Shmop $shmId): never {
    // Process enfant (fork)
    echo "Je suis le \033[32mfork\033[0m, surveillance de la mémoire partagée..." . PHP_EOL;

    $lastValue = '';
    $running = true;

    // 3 - Lire le contenu de la mémoire partagée en boucle
    while ($running) {
        $currentValue = trim(shmop_read($shmId, 0, 10));

        if ($currentValue !== $lastValue && $currentValue !== '0') {
            echo "Fork détecte un changement: \033[32m$currentValue\033[0m" . PHP_EOL;
            $lastValue = $currentValue;

            // Arrêter après avoir détecté la valeur 3
            if ($currentValue === '3') {
                $running = false;
            }
        }

        usleep(50_000); // 50ms
    }

    echo "\033[32mFork terminé\033[0m" . PHP_EOL;
    exit(0);
}
```

Voici une fonction qui s’executera dans notre fork, on lira en boucle un emplacement mémoire et lorsqu’un changement sera détecté, on va afficher la nouvelle valeur. Pour éviter une busy loop on attendra 50 ms entre 2 lectures. Puis quand on aura tout lu, on arrêtera le fork.

Voici désormais ce qu’on exécutera dans le processus principal. Toute les 100ms nous allons incrémenter la valeur présente dans la mémoire partagée, 3 fois de suite. Puis nous attendrons la fin de l’exécution du processus enfant avant de détruire l’emplacement mémoire.

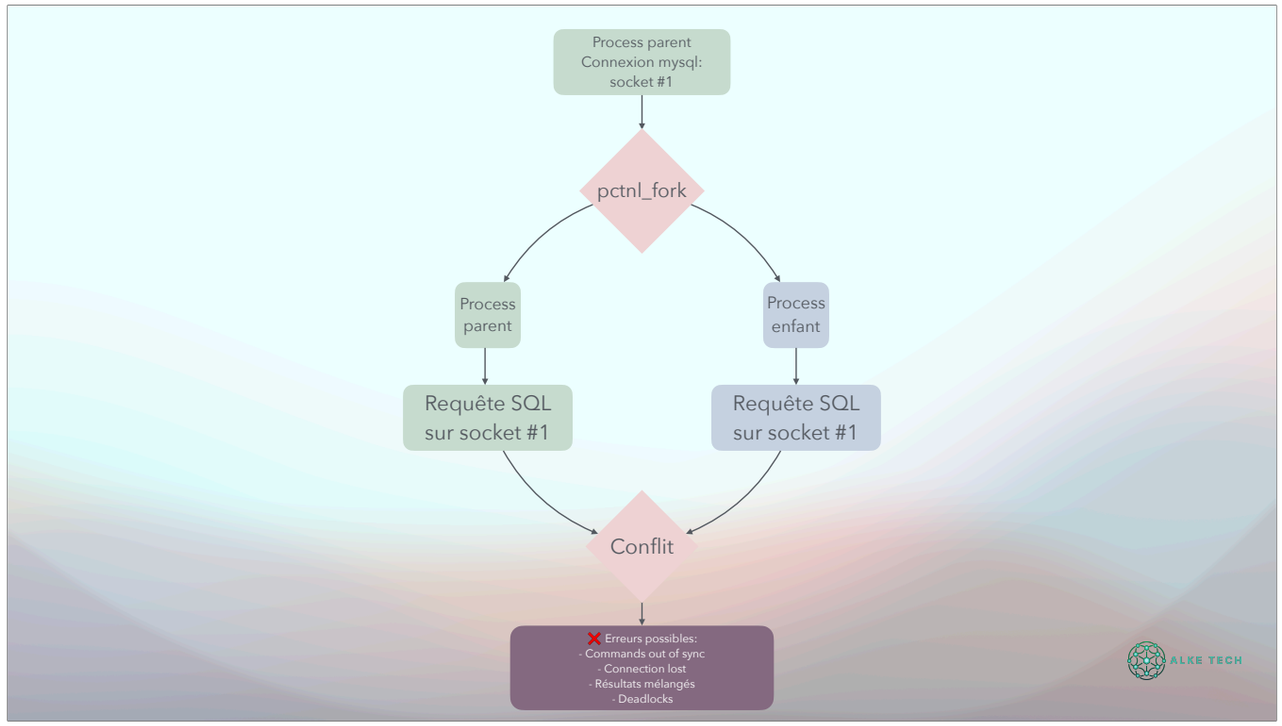

```php
$shmKey = ftok(__FILE__, 't');
$shmId = shmop_open($shmKey, 'c', 0644, 1024);

if ($shmId === false) {
    die("\033[31mErreur: Impossible de créer la mémoire partagée\033[0m" . PHP_EOL);
}

shmop_write($shmId, str_pad('0', 10, ' '), 0);

$pid = pcntl_fork();

if ($pid === 0) {
    executeInFork($shmId);
} elseif ($pid === -1) {
    echo "\033[31mErreur: Impossible de forker\033[0m" . PHP_EOL;
    shmop_delete($shmId);
} else {
    executeInMainThread($shmId);
}
```

Et pour assembler le tout, voici ce que ça donne:

1. On crée une clé unique pour notre emplacement mémoire à l'aide la fonction ftok. D'autres options sont possibles mais c'est généralement la manière recommandée
2. On crée l'emplacement mémoire
3. On initialise à « 0 » + vide le contenu de cet emplacement
4. On fork les process et on voit ce que ça donne

Démonstration: php exemples/step6-shmop.php

Est-ce que vous avez remarqué que l'ouverture vers la mémoire partagée n'a été faite qu'une seule fois puis partagée entre le processus enfant et le fork ?
Ici on tombe dans un des gros points d'attention de l'utilisation des forks: les ressources externes.



En effet, tous les pointeurs (vers des connexions maintenues par des libs externes) étant copiés lorsqu'on fork le process, on peut avoir des surprises inattendues.
Par exemple avec mysql (mais le problème va être le même un peu partout):

1. J'ai un script qui a une connexion mysql ouverte, imaginons que la socket #1 soit utilisée par le système pour cette connexion
2. On réalise le fork comme vu précédemment
3. On a ensuite 2 process qui coexistent, l'enfant étant relié au parent
4. Tous les 2 cherchent à exécuter une requête sql, sur la même socket #1 qu'au début
5. Le driver ne le sait pas à l'avance, mais il va y avoir un conflit avec différentes erreurs possibles

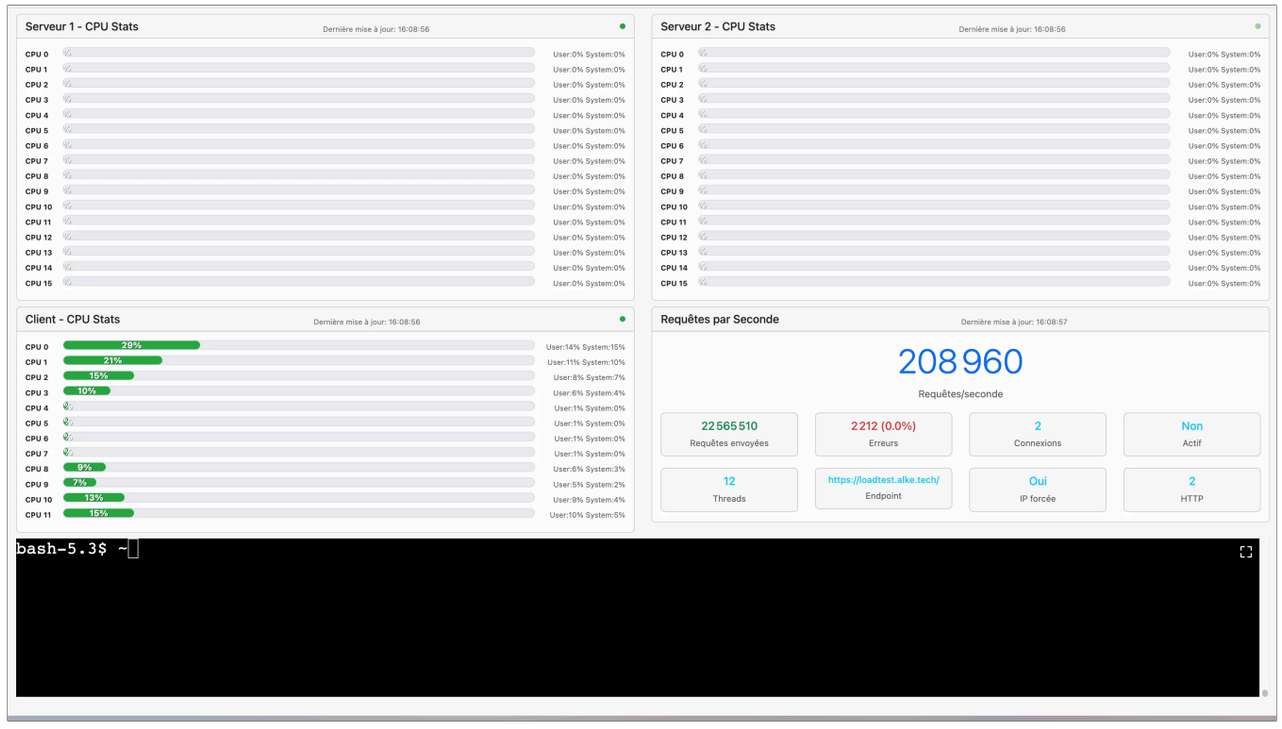
7. Et là en général c'est le bazar, bon courage pour rattraper le coup

Pour éviter cela, plusieurs options:

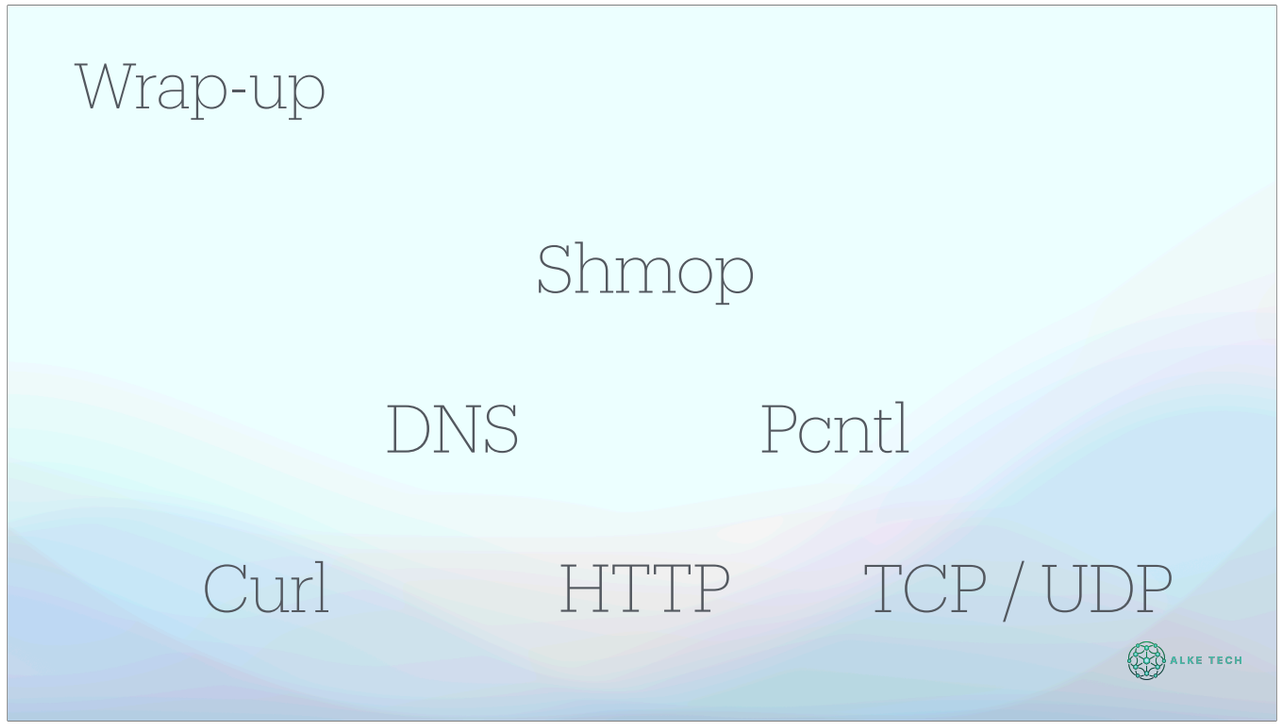
1. N'établir des connexions qu'après la création du fork
2. Si ça n'est pas possible, forcer des reconnexions (par exemple à l'aide d'une méthode ping)

Maintenant que nous avons toutes les bases, voyons voir ce que ça donne !

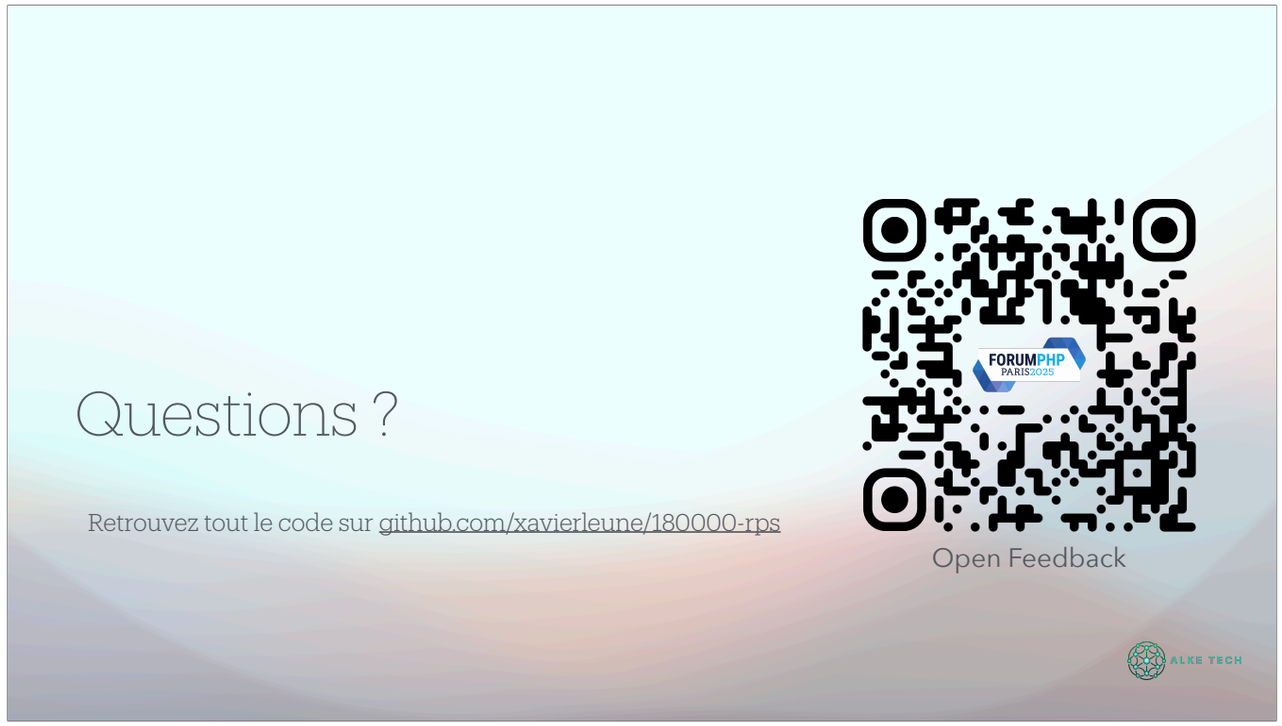
`bin/console app:notification-sender 12 --force-hostname-resolution --env=prod --no-debug`



THREAD: 12
DNS: forcé
HTTP: 2



Au cours de ce talk, nous avons pu rentrer dans les détails de curl, parler de l’évolution de HTTP, de protocoles bas niveau tels que TCP et UDP, de résolution DNS et enfin de forks et de mémoire partagée.
J’espère que ce talk vous a plu et que vous y avez appris des choses



Tout le code que vous avez vu aujourd’hui et même celui que vous n’avez pas vu est dispo sur github.
Avez-vous des questions ?