nginx

(Linux 2.6 edition)

nicolas.bazire@gmail.com
06-12-2011

Scalable event-driven asynchronous non-blocking wonder

Au menu

nginx : son histoire, ses fonctions

Concepts fondateurs, Architecture

Comment ça marche sur Linux 2.6 ? (enfin, en partie)

Pourquoi c'est le bien (ou pas)

WTF is nginx!?

Daemon réseau écrit en C :

- Serveur HTTP (contenu statique, *CGI, streaming)
- Reverse proxy HTTP/Mail (+ cache)
- HTTP load balancer
- SSL accelerator

Il était une fois...

Créé par **Igor Sysoev** en 2002, administrateur système senior chez Rambler Media Group

Problème à l'époque :
encaisser la montée en charge causée par la hausse des connexions concurrentes (clients lents)

De nos jours

Le problème reste entier, mais les causes sont différentes :

- Connexions persistantes (keep-alive)
- Internet mobile
- Navigateurs modernes (requêtes concurrentes)

Avant d'aller plus loin

Scalability != Performance

Scalability :

> Je sers 100 req/sec avec 1000 connexions concurrentes, comment faire de même avec 10,000 connexions?

Performance :

> Je sers 100 req/sec avec 1000 connexions concurrentes, comment faire pour servir 500 req/sec?

Concepts fondateurs

Axé sur la concurrence

Utilisation très réduite de la mémoire

Asynchronous non-blocking I/O

Event-driven model

Économe sur les appels système

Architecture

1 exécutable :

N processus “worker” :

Traite les requêtes clients
Gère les connexions

1 processus “master” :

Surveille les workers
Gère les signaux (HUP, INT, TERM, … WINCH)

Event-driven model

master process :

- sigsuspend();
  - HUP : Recharger le fichier de configuration
  - USR2 : Changer de binaire
  - TERM : Shutdown
  - etc.

Event-driven model
+ Asynchronous non-blocking I/O

worker process :

```
epoll_create(); // renvoie une instance d'epoll (fd)
epoll_ctl();    // contrôle l'instance epoll
epoll_wait();   // indique les évènements en attente
accept4();      // SOCK_NONBLOCK
open();         // O_NONBLOCK
```

Les Quatre Cavaliers de l'Apocalypse

Copie de données

Allocation mémoire

Changement de contexte

Lock contention

Copie de données
Allocation mémoire

Utiliser des descripteurs de buffer chaînés

Attention aux copies cachées
(kernel buffer != user buffer != NIC buffer)

→ sendfile(); // transfert direct entre 2 fd

Memory pool (+ pseudo garbage collector)

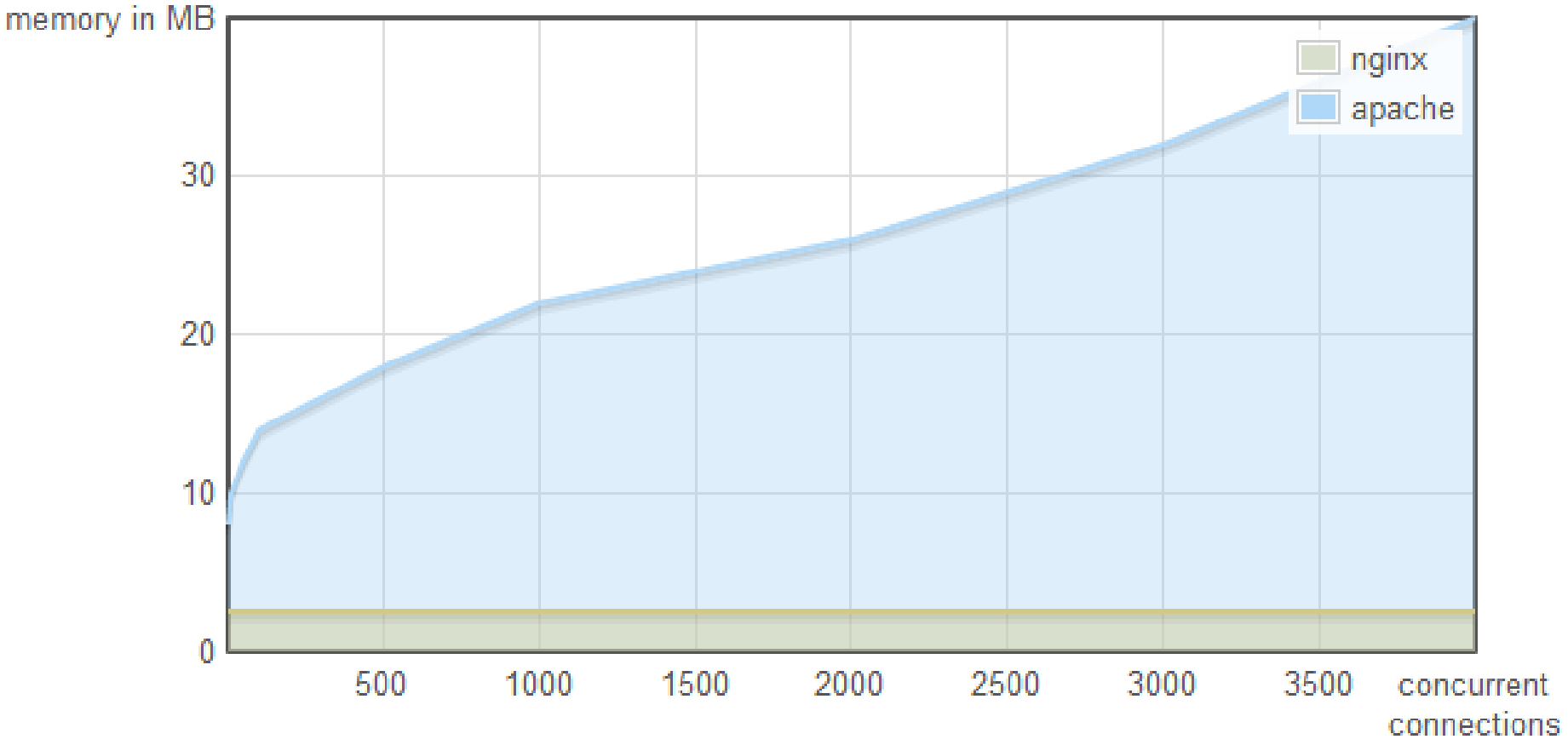
Changement de contexte
Lock contention

Pas de threads!

Un client est traité par un seul et unique worker du début à la fin : pas de lock contention, pas de changement de contexte forcé

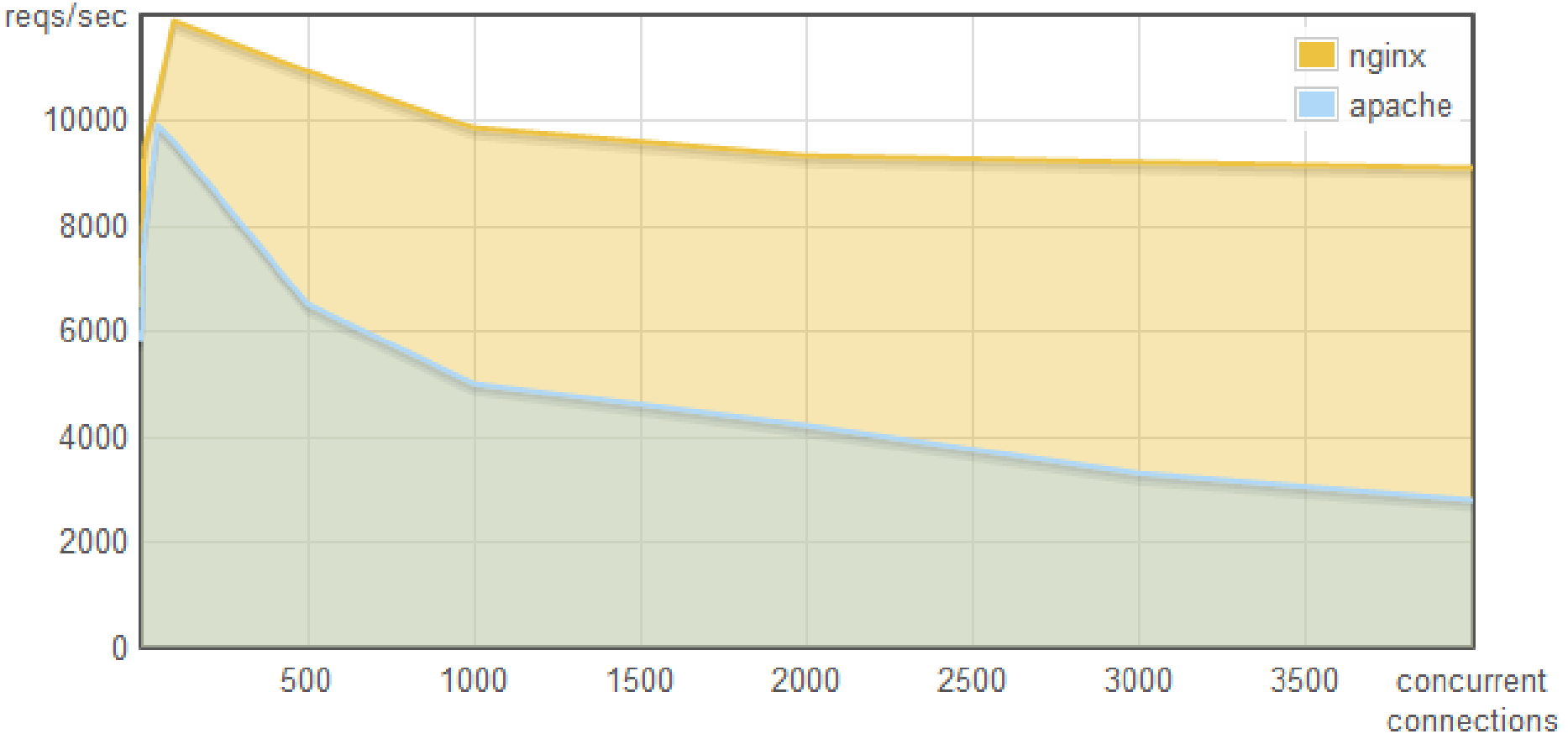
Context switch != Mode switch

Consommation mémoire



src: webfaction

Throughput



src: webfaction

Stats

	Sites actifs	Part
Apache	100,516,026	58.36%
IIS	21,109,639	12.26%
nginx	19,982,947	11.60%
autres	30,391,388	17.78%
	172,000,000	

src: netcraft (10/2011)

Stats

	Top 1,000,000	Top 100,000	Top 10,000	Top 1000
Apache	67%	58.9%	50.9%	43.2%
IIS	18.4%	19.2%	17.1%	13.4%
nginx	9.8%	15.2%	21.6%	24.8%

src: w3techs (12/2011)

Success stories

sourceforge.net, dropbox.com, github.com, groupon.com... youporn.com :D

wordpress.com (25 millions de blogs) : load-balancer → frontend

wikipedia.org : SSL accelerator

Pourquoi c'est le bien

Pas obligatoire de se servir de tous les modules

Dépendances de librairies facultatives :

- OpenSSL
- zlib (gzip)
- md5/sha1 (adressage)
- PCRE (URL rewriting)

Très peu gourmand en mémoire

Conçu pour gérer plusieurs milliers de connexions simultanées

!one-size-fits-all

Pas de chargement dynamique de modules

Pas d'URL rewriting à la volée (.htaccess)

Pas d'exécution de langage dynamique (mod_php)

Implique de repenser son architecture

Mauvais choix pour une machine mutualisée!


Questions?