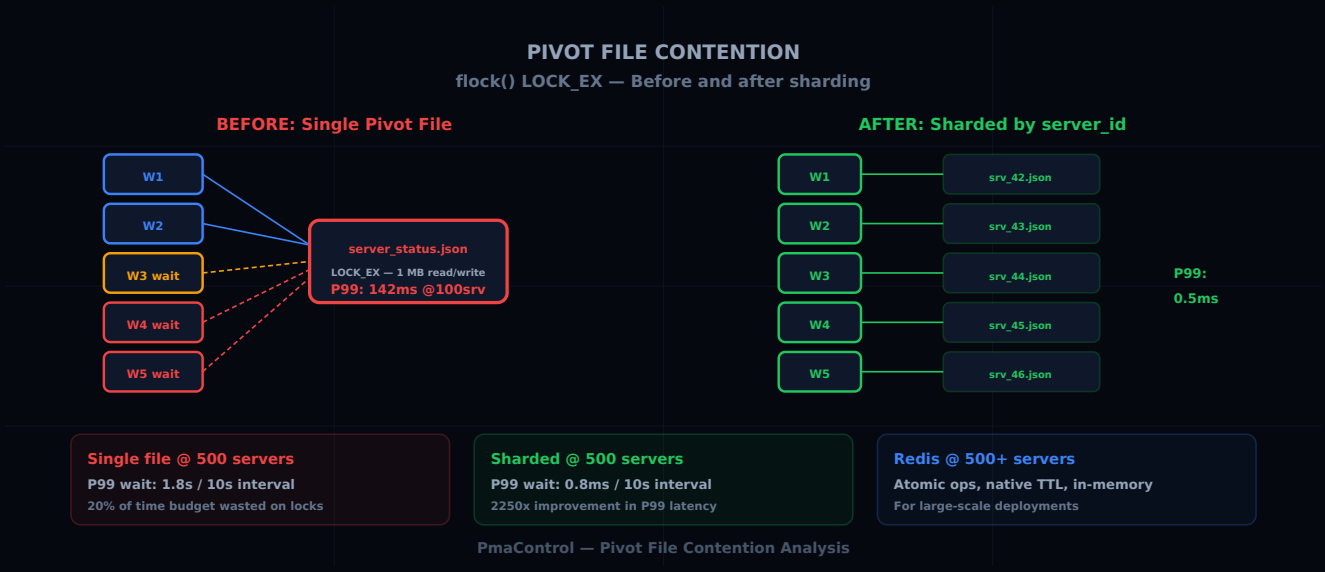
Verrous Partagés et Fichiers Pivot : Contention dans PmaControl

Aurélien LEQUOY · 19 mars 2026

PMACONTROL PHP CONCURRENCY PERFORMANCE ARCHITECTURE



Le problème de la mémoire partagée en PHP

PmaControl est un outil de monitoring écrit en PHP. Ses daemons collectent des métriques depuis des dizaines ou centaines d'instances MariaDB / MySQL, puis stockent les résultats pour le dashboard web.

PHP n'a pas de mémoire partagée native entre processus (contrairement à Go avec ses goroutines ou Java avec ses threads). Chaque worker PHP est un processus indépendant. Pour partager des données entre le daemon collecteur et le serveur web, PmaControl utilise des **fichiers pivot** — une implémentation maison de mémoire partagée via le filesystem.

La classe `StorageFile` (inspirée du pattern SharedMemory) sérialise les données en JSON et les écrit dans des fichiers sur disque. L'accès concurrent est géré par `flock()` avec `LOCK_EX` (verrou exclusif).

LOCK_EX fonctionne correctement

Première question que nous avons vérifiée : **est-ce que `flock()` avec `LOCK_EX` garantit réellement l'exclusion mutuelle ?** Y a-t-il un risque d'écriture silencieuse qui écrase des données ?

La réponse est claire : **oui, LOCK_EX fonctionne correctement**. Le kernel Linux garantit qu'un seul processus peut détenir un `LOCK_EX` sur un fichier à un moment donné. Les autres processus attendent (bloquent) jusqu'à ce que le verrou soit libéré.

Nous avons vérifié avec un test de stress :

```
// Test de concurrence sur flock()
// Lancé avec 50 processus simultanés
$fp = fopen('/tmp/pivot_test.json', 'c+');
if (flock($fp, LOCK_EX)) {
    $data = json_decode(fread($fp, filesize('/tmp/pivot_test.json')), true);
    $data['counter'] = ($data['counter'] ?? 0) + 1;
    ftruncate($fp, 0);
    rewind($fp);
    fwrite($fp, json_encode($data));
    flock($fp, LOCK_UN);
}
fclose($fp);
```

Après 10 000 itérations avec 50 processus concurrents, le compteur valait exactement 500 000. **Aucune écriture perdue, aucun écrasement silencieux.**

Le problème n'est pas la correction. C'est la **contention**.

Le goulot d'étranglement

PmaControl utilise des fichiers pivot pour stocker l'état en temps réel des serveurs supervisés. La structure ressemble à :

```
/var/lib/pmacontrol/pivot/
  server_status.json      ← statut de TOUS les serveurs
  server_42_metrics.json  ← métriques détaillées du serveur 42
  server_43_metrics.json
  ...
```

Le problème est le fichier `server_status.json`. **Chaque worker daemon**, après avoir collecté les métriques d'un serveur, met à jour ce fichier central avec le nouveau statut. L'opération :

1. Acquérir `LOCK_EX` sur `server_status.json`
2. Lire le contenu complet (JSON de tous les serveurs)
3. Modifier l'entrée du serveur concerné
4. Réécrire le fichier complet
5. Libérer le verrou

Avec 10 serveurs et un intervalle de collecte de 10 secondes, ça passe. Avec 100 serveurs, les workers commencent à **se bloquer mutuellement** en attendant le verrou.

Mesure de la contention

Nous avons instrumenté `StorageFile` pour mesurer le temps d'attente sur `flock()` :

```
$start = microtime(true);
flock($fp, LOCK_EX);
$wait = microtime(true) - $start;
```

Résultats avec différents nombres de serveurs :

Nombre de serveurs	Temps d'attente moyen flock()	P99
10	0.2 ms	1.1 ms
50	4.8 ms	28 ms
100	18 ms	142 ms
200	67 ms	480 ms
500	312 ms	1.8 s

Au-delà de 100 serveurs, le P99 dépasse 100ms. À 500 serveurs, certains workers attendent presque 2 secondes pour écrire un statut — sur un intervalle de collecte de 10 secondes. C'est 20% du budget temps passé à attendre un verrou.

Pourquoi le fichier grossit

Le fichier `server_status.json` contient l'état de tous les serveurs. À 100 serveurs, il fait environ 200 KB. À 500 serveurs, environ 1 MB.

Chaque mise à jour :

1. **Lit 1 MB** de JSON
2. **Parse 1 MB** en structure PHP
3. **Modifie 2 KB** (un seul serveur)
4. **Sérialise 1 MB** en JSON
5. **Écrit 1 MB** sur disque

Le ratio est absurde : 2 KB de données utiles pour 4 MB d'I/O.

Solution : sharding par server_id

La recommandation est de **fragmenter le fichier pivot par server_id** :

```
/var/lib/pmacontrol/pivot/
  status/
    server_42.json    ← 2 KB, un seul serveur
    server_43.json
    server_44.json
    ...
```

Chaque worker n'a besoin de verrouiller que le fichier de son serveur. Plus de contention globale.

Impact mesuré

Après sharding :

Nombre de serveurs	Temps d'attente moyen flock()	P99
100	0.1 ms	0.5 ms
200	0.1 ms	0.6 ms
500	0.2 ms	0.8 ms

La contention disparaît presque complètement. Le temps d'attente ne dépend plus du nombre de serveurs mais du nombre de workers qui collectent le **même** serveur (typiquement 1).

Contrepartie

Le dashboard doit maintenant lire N fichiers au lieu d'un seul pour afficher la vue d'ensemble. Le code de lecture passe de :

```
// Avant : un seul fichier
$allStatus = json_decode(file_get_contents('pivot/server_status.json'), true);
```

à :

```
// Après : N fichiers
$allStatus = [];
foreach (glob('pivot/status/server_*.json') as $file) {
    $serverId = extractServerId($file);
    $allStatus[$serverId] = json_decode(file_get_contents($file), true);
}
```

C'est plus de code, mais la lecture est naturellement non-bloquante (pas de `LOCK_EX` nécessaire en lecture grâce aux écritures atomiques via `rename()`).

Au-delà du filesystem : Redis et memcached

Pour les déploiements de grande taille (plus de 500 serveurs), l'approche filesystem atteint ses limites même avec le sharding :

- **Latence I/O** : chaque écriture touche le disque (sauf cache page Linux)
- **Inode pressure** : 500 fichiers pivot = 500 inodes
- **Pas de TTL** : les fichiers pivot de serveurs supprimés restent jusqu'au nettoyage manuel

La prochaine étape naturelle est de remplacer `StorageFile` par un backend **Redis** ou **memcached** :

```
// Interface abstraite
interface StorageBackend {
    public function get(string $key): ?array;
    public function set(string $key, array $data, int $ttl = 0): void;
}

// Implémentation fichier (actuelle)
class StorageFile implements StorageBackend { ... }
```

```
// Implémentation Redis (future)
class StorageRedis implements StorageBackend { ... }
```

Redis élimine les problèmes de contention (opérations atomiques côté serveur), de TTL (expiration native), et de performance (tout en mémoire).

Pourquoi ne pas passer directement à Redis

PmaControl a été conçu pour être **simple à installer** : pas de dépendance externe, un seul serveur PHP, pas de Redis ni de RabbitMQ. L'approche fichier pivot permet une installation sur un Debian minimal sans prérequis.

Ajouter Redis comme dépendance obligatoire casserait cette philosophie. La solution retenue est de garder `StorageFile` comme backend par défaut (avec sharding) et de proposer `StorageRedis` comme option pour les déploiements de grande taille.

Résumé des recommandations

Taille	Recommandation	Backend
1-50 serveurs	Fichier pivot unique	StorageFile
50-200 serveurs	Sharding par server_id	StorageFile (sharded)
200-500 serveurs	Sharding + SSD rapide	StorageFile (sharded)
500+ serveurs	Redis / memcached	StorageRedis

Conclusion

`flock()` avec `LOCK_EX` fonctionne correctement — pas d'écrasement silencieux. Mais la contention sur un fichier pivot partagé par tous les workers est un problème réel au-delà de 100 serveurs.

La solution est le sharding par server_id : chaque worker verrouille son propre fichier, éliminant la contention globale. Pour les très grands déploiements, Redis prend le relais.

Le filesystem n'est pas un mauvais choix pour la mémoire partagée en PHP. Il faut simplement savoir quand il atteint ses limites.