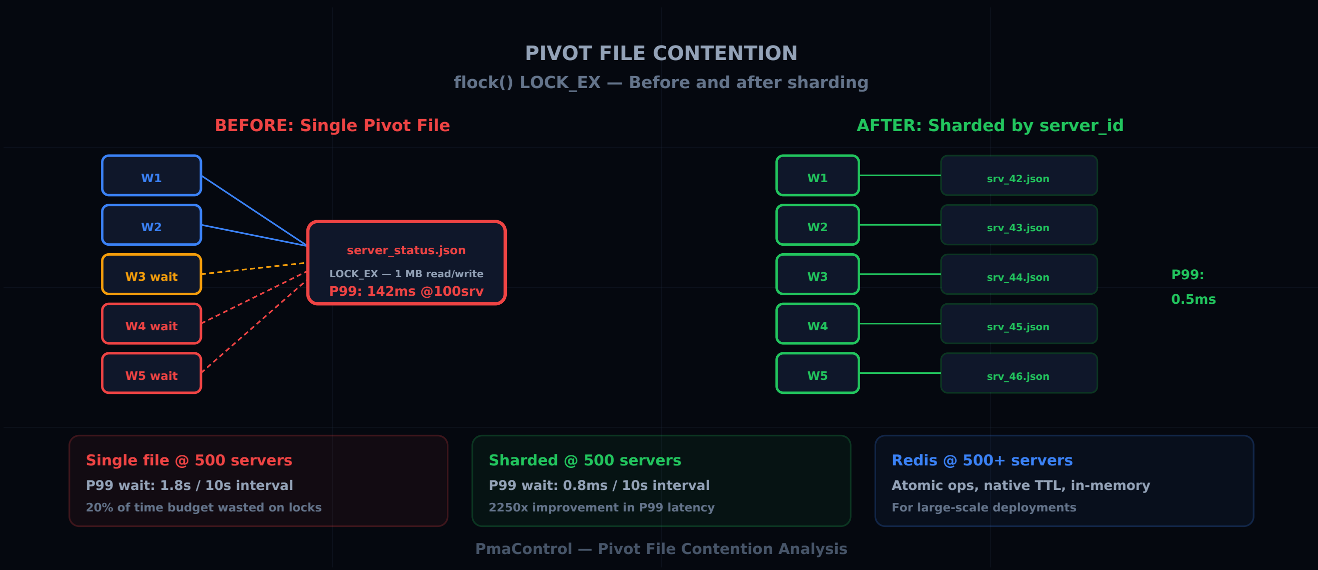
[illegible]PmaControl [illegible]

Aurélien LEQUOY · 2026[illegible]3[illegible]19[illegible]

PMACONTROL · PHP · CONCURRENCY · PERFORMANCE · ARCHITECTURE



PHP [illegible]

PmaControl [illegible] PHP [illegible] MariaDB / MySQL [illegible] Web [illegible]

PHP [illegible] Go [illegible] goroutine [illegible] Java [illegible] PHP [illegible] **pivot files** [illegible] Web [illegible] PmaCo[illegible]

`StorageFile` [illegible] SharedMemory [illegible] `flock()` [illegible] `LOCK_EX` [illegible]

LOCK_EX [illegible]

`flock()` [illegible] `LOCK_EX` [illegible]

[illegible] **LOCK_EX** [illegible] Linux [illegible] `LOCK_EX` [illegible]

[illegible]

```
// 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);
```

在 50 个并发进程下运行 10,000 次迭代后，计数器的值精确为 500,000。

问题不在于正确性，而在于锁竞争。

瓶颈所在

PmaControl 使用中枢文件来存储被监控服务器的实时状态。目录结构如下：

```
/var/lib/pmacontrol/pivot/
  server_status.json      ← 所有服务器的状态
  server_42_metrics.json  ← 服务器 42 的详细指标
  server_43_metrics.json
  ...
```

问`server_status.json` 文件上。每个守护进程 **worker**台服务器的指标后，都会用新状态更新这个中心文件。操作流程：

1. 获`server_status.json` 的`LOCK_EX`
2. 读取完整内容（包含所有服务器的 JSON）
3. 修改对应服务器的条目
4. 重写整个文件
5. 释放锁

在 10 台服务器、采集间隔 10 秒的情况下，这没有问题。但到了 100 台服务器时，worker 们开始

锁竞争测量

我`StorageFile` 进`flock()`量 的等待时间：

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

不同服务器数量下的测量结果:

服务器数量	**flock()** 平均等待时间	**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

超过 100 台服务器后，P99 超过 100ms。在 500 台服务器时，某些 worker 需要等待近 2 秒才能写入状态——而采集间隔只有 10 秒。这意味着 20% 的时间预算花在了等待锁上。

文件为何不断增大

`server_status.json` 文件包含所有服务器的状态。100 台服务器时约 200 KB，500 台服务器时约 1 MB。

每次更新:

1. 读取 **1 MB** JSON
2. 解析 **1 MB** PHP 数据结构
3. 修改 **2 KB** (一台服务器)
4. 序列化 **1 MB** JSON
5. 写入 **1 MB** 磁盘

这个比率荒谬至极: 2 KB 的有效数据却产生 4 MB 的 I/O。

解决方案: 按 **server_id** 分片

建议按 **server_id** 对中枢文件进行分片:

```
/var/lib/pmacontrol/pivot/
  status/
    server_42.json    ← 2 KB, 仅一台服务器
    server_43.json
    server_44.json
    ...
```

每个 worker 只需要锁定自己负责的服务器文件，全局锁竞争不复存在。

测量结果

分片后：

服务器数量	**flock()** 平均等待时间	**P99**
100	0.1 ms	0.5 ms
200	0.1 ms	0.6 ms
500	0.2 ms	0.8 ms

[illegible] worke[illegible]待数量不[illegible]取决于[illegible]务器数量，而是取决于采集

代价

仪表盘现在需要读取 N 个文件而不是一个文件来显示全局视图。读取代码从：

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

变为：

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

代 rename (读取[illegible]写入([illegible]不需要

超越文件系统：**Redis** 和 **memcached**

对于大规模部署（超过 500 台服务器），即使使用了分片，文件系统方案也会达到极限：

- **I/O** 延迟：每次写入都触及磁盘（除非有 Linux 页面缓存）
- **Inode** 压 500 个中枢文件 = 500 个 inode
- 没有 **TTL**：删除服务器的中枢文件会一直保留，直到手动清理

下`St`自然的演进`File` **Redis** **memcached** 后端:

```
// 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 消除了锁竞争问题（服务端原子操作）、TTL 问题（原生过期机制）和性能问题（全内存操作）。

为什么不直接切换到 **Redis**

PmaControl 的设计理念是安装简单：无外部依赖，只需一台 PHP 服务器，不需要 Redis 或 RabbitMQ。中枢文件方案使得在最小化安装的 Debian 上无需任何前置条件即可部署。

将 Redis `St`强制依赖会`Fil`—设计[illegible]

建议总结

规模	建议	后端
1-50 台服务器	单一中枢文件	StorageFile
50-200 台服务器	按 server_id 分片	StorageFile（分片）
200-500 台服务器	分片 + 高速 SSD	StorageFile（分片）
500+ 台服务器	Redis / memcached	StorageRedis

结论

`flock()` 配合OCK_EX 工作正常——不会发生静默覆写。但当所有 worker 共享同一个中枢文件时，锁竞争在超过 100 台服务器后就成为真实的问题。

解决方案是按 server_id 分片：每个 worker 锁定自己的文件，消除全局锁竞争。对于超大规模部署，Redis 接替文件系统。

文件系统并非 PHP 共享内存的糟糕选择，只需要清楚它在何时会达到极限。