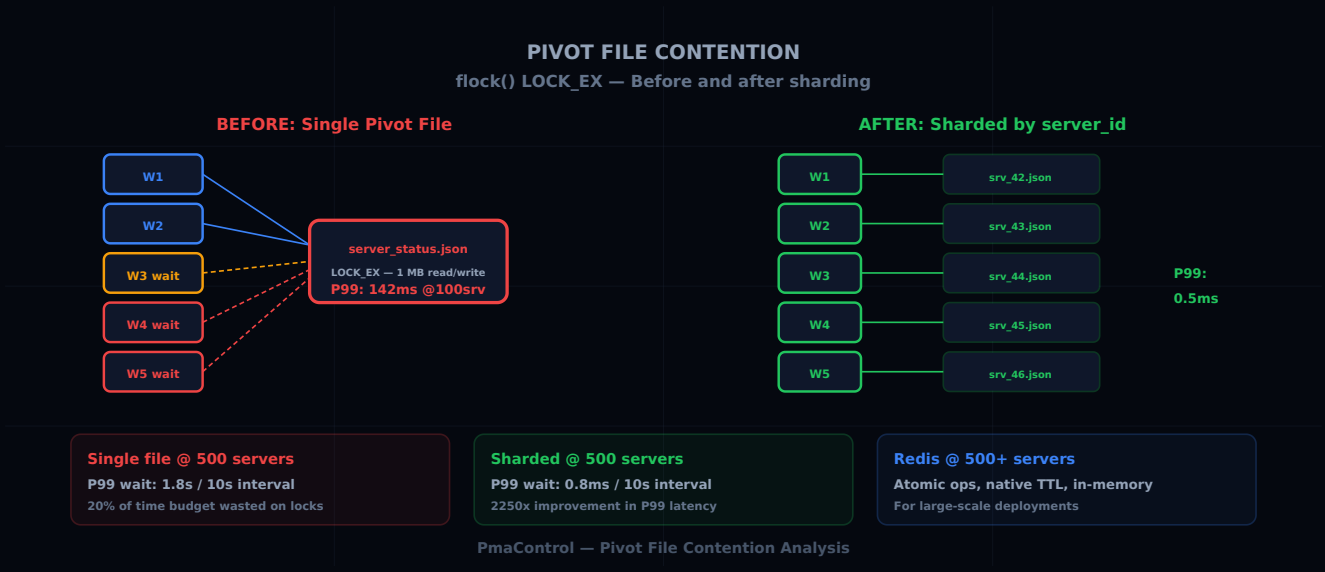
Blokady współdzielone i pliki pivot: rywalizacja w PmaControl

Aurélien LEQUOY · March 19, 2026

PMACONTROL PHP CONCURRENCY PERFORMANCE ARCHITECTURE



Problem pamięci współdzielonej w PHP

PmaControl to narzędzie monitoringu napisane w PHP. Jego demony zbierają metryki z dziesiątek lub setek instancji MariaDB / MySQL, a następnie przechowują wyniki dla dashboardu webowego.

PHP nie posiada natywnej pamięci współdzielonej między procesami (w przeciwieństwie do Go z goroutines czy Javy z wątkami). Każdy worker PHP jest niezależnym procesem. Aby dzielić dane między demonem zbierającym a serwerem WWW, PmaControl używa **plików pivot** — własnej implementacji pamięci współdzielonej za pośrednictwem systemu plików.

Klasa `StorageFile` (inspirowana wzorcem SharedMemory) serializuje dane do JSON i zapisuje je w plikach na dysku. Współbieżny dostęp jest zarządzany przez `flock()` z `LOCK_EX` (blokada wyłączna).

LOCK_EX działa poprawnie

Pierwszym pytaniem, które zweryfikowaliśmy, było: **czy `flock()` z `LOCK_EX` rzeczywiście gwarantuje wzajemne wykluczenie?** Czy istnieje ryzyko cichego zapisu nadpisującego dane?

Odpowiedź jest jednoznaczna: **tak, LOCK_EX działa poprawnie**. Jądro Linuxa gwarantuje, że tylko jeden proces może posiadać `LOCK_EX` na pliku w danym momencie. Pozostałe procesy czekają (blokują się), dopóki blokada nie zostanie zwolniona.

Zweryfikowaliśmy to testem obciążeniowym:

```
// Test współbieżności flock()
// Uruchomiony z 50 jednoczesnych procesów
$fp = fopen('/tmp/pivot_test.json', 'c+');
if (flock($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);
```

Po 10 000 iteracjach z 50 współbieżnymi procesami licznik wynosił dokładnie 500 000. **Żaden zapis nie został utracony, żadne ciche nadpisanie.**

Problem nie leży w poprawności. To **rywalizacja**.

Wąskie gardło

PmaControl używa plików pivot do przechowywania stanu w czasie rzeczywistym nadzorowanych serwerów. Struktura wygląda tak:

```
/var/lib/pmacontrol/pivot/
  server_status.json      <- status WSZYSTKICH serwerów
  server_42_metrics.json  <- szczegółowe metryki serwera 42
  server_43_metrics.json
  ...
```

Problemem jest plik `server_status.json`. **Każdy worker demona**, po zebraniu metryk serwera, aktualizuje ten centralny plik nowym statusem. Operacja:

1. Uzyskanie `LOCK_EX` na `server_status.json`
2. Odczyt całej zawartości (JSON wszystkich serwerów)
3. Modyfikacja wpisu danego serwera
4. Ponowny zapis całego pliku
5. Zwolnienie blokady

Przy 10 serwerach i interwale zbierania 10 sekund to działa. Przy 100 serwerach workery zaczynają **wzajemnie się blokować** czekając na blokadę.

Pomiar rywalizacji

Zinstrumentowaliśmy `StorageFile`, aby mierzyć czas oczekiwania na `flock()`:

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

Wyniki dla różnej liczby serwerów:

Liczba serwerów	Średni czas oczekiwania 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

Powyżej 100 serwerów P99 przekracza 100 ms. Przy 500 serwerach niektóre workery czekają prawie 2 sekundy na zapis statusu — przy interwale zbierania 10 sekund. To 20% budżetu czasu spędzonego na czekaniu na blokadę.

Dlaczego plik rośnie

Plik `server_status.json` zawiera stan wszystkich serwerów. Przy 100 serwerach zajmuje około 200 KB. Przy 500 serwerach — około 1 MB.

Każda aktualizacja:

1. **Odczytuje 1 MB** JSON
2. **Parsuje 1 MB** do struktury PHP
3. **Modyfikuje 2 KB** (jeden serwer)
4. **Serializuje 1 MB** do JSON
5. **Zapisuje 1 MB** na dysk

Proporcja jest absurdalna: 2 KB użytecznych danych na 4 MB operacji I/O.

Rozwiązanie: sharding po server_id

Rekomendacją jest **fragmentacja pliku pivot po server_id**:

```
/var/lib/pmacontrol/pivot/
  status/
    server_42.json    <- 2 KB, jeden serwer
    server_43.json
    server_44.json
    ...
```

Każdy worker musi zablokować tylko plik swojego serwera. Koniec z globalną rywalizacją.

Zmierzony wpływ

Po shardingu:

Liczba serwerów	Średni czas oczekiwania flock()	P99
100	0.1 ms	0.5 ms
200	0.1 ms	0.6 ms
500	0.2 ms	0.8 ms

Rywalizacja prawie całkowicie znika. Czas oczekiwania nie zależy już od liczby serwerów, lecz od liczby workerów zbierających dane z **tego samego** serwera (typowo 1).

Kompromis

Dashboard musi teraz odczytać N plików zamiast jednego, aby wyświetlić widok ogólny. Kod odczytu zmienia się z:

```
// Przed: jeden plik
$allStatus = json_decode(file_get_contents('pivot/server_status.json'), true);
```

na:

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

To więcej kodu, ale odczyt jest naturalnie nieblokujący (nie potrzeba `LOCK_EX` przy odczycie dzięki atomowym zapisom przez `rename()`).

Poza systemem plików: Redis i memcached

Dla wdrożeń dużej skali (powyżej 500 serwerów) podejście z systemem plików osiąga swoje limity nawet z shardingiem:

- **Latencja I/O**: każdy zapis dotyka dysku (z wyjątkiem cache stron Linuxa)
- **Obciążenie inode**: 500 plików pivot = 500 inode'ów
- **Brak TTL**: pliki pivot usuniętych serwerów pozostają do ręcznego czyszczenia

Naturalnym kolejnym krokiem jest zastąpienie `StorageFile` backendem **Redis** lub **memcached**:

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

// Implementacja plikowa (obecna)
class StorageFile implements StorageBackend { ... }

// Implementacja Redis (przyszła)
class StorageRedis implements StorageBackend { ... }
```

Redis eliminuje problemy rywalizacji (operacje atomowe po stronie serwera), TTL (natywne wygasanie) oraz wydajności (wszystko w pamięci).

Dlaczego nie przejść od razu na Redis

PmaControl został zaprojektowany tak, aby był **prosty w instalacji**: bez zewnętrznych zależności, pojedynczy serwer PHP, bez Redisa ani RabbitMQ. Podejście z plikami pivot pozwala na instalację na minimalnym Debianie bez wymagań wstępnych.

Dodanie Redisa jako obowiązkowej zależności złamałoby tę filozofię. Przyjęte rozwiązanie to zachowanie `StorageFile` jako domyślnego backendu (z shardingiem) i zaproponowanie `StorageRedis` jako opcji dla wdrożeń dużej skali.

Podsumowanie rekomendacji

Rozmiar	Rekomendacja	Backend
1-50 serwerów	Pojedynczy plik pivot	StorageFile
50-200 serwerów	Sharding po server_id	StorageFile (sharded)
200-500 serwerów	Sharding + szybkie SSD	StorageFile (sharded)
500+ serwerów	Redis / memcached	StorageRedis

Podsumowanie

`flock()` z `LOCK_EX` działa poprawnie — żadnych cichych nadpisań. Ale rywalizacja na współdzielonym pliku pivot przez wszystkie workery to realny problem powyżej 100 serwerów.

Rozwiązaniem jest sharding po server_id: każdy worker blokuje swój własny plik, eliminując globalną rywalizację. Dla bardzo dużych wdrożeń Redis przejmuje pałeczkę.

System plików nie jest złym wyborem dla pamięci współdzielonej w PHP. Trzeba po prostu wiedzieć, kiedy osiąga swoje limity.