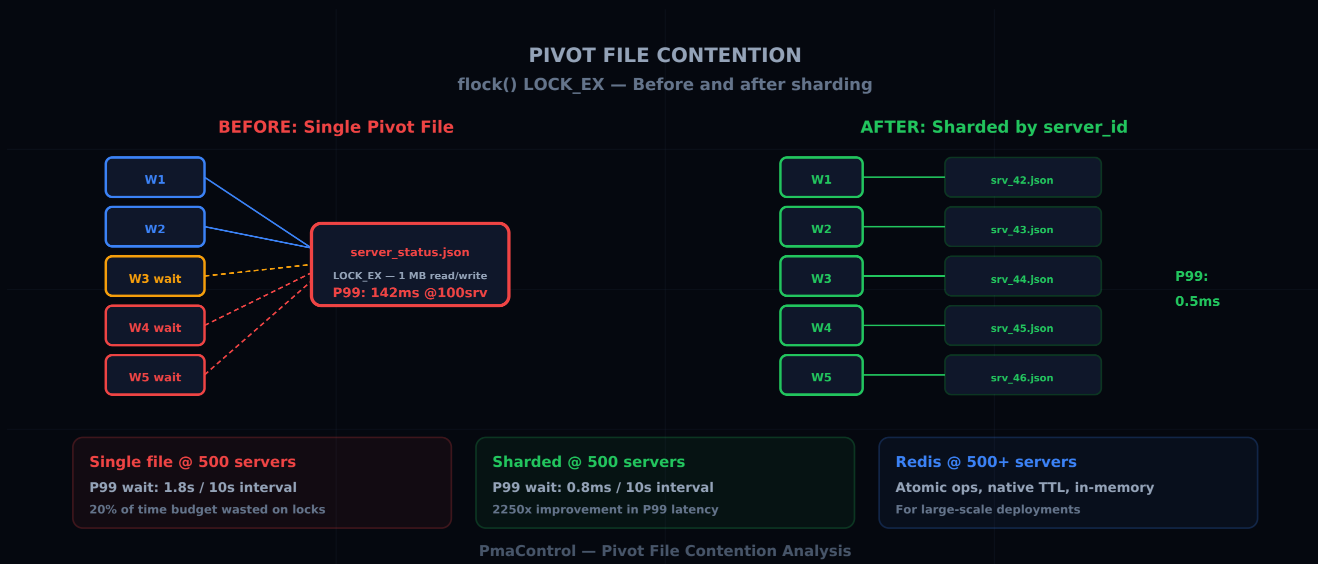
Разделяемые блокировки и файлы-пивоты: контеншн в PmaControl

Aurélien LEQUOY · March 19, 2026

PMACONTROL PHP CONCURRENCY PERFORMANCE ARCHITECTURE



Проблема разделяемой памяти в PHP

PmaControl — инструмент мониторинга, написанный на PHP. Его демоны собирают метрики с десятков или сотен экземпляров MariaDB / MySQL, затем сохраняют результаты для веб-дашборда.

PHP не имеет нативной разделяемой памяти между процессами (в отличие от Go с его горутинами или Java с потоками). Каждый PHP-воркер — независимый процесс. Для обмена данными между демоном сбора и веб-сервером PmaControl использует **файлы-пивоты** — собственную реализацию разделяемой памяти через файловую систему.

Класс `StorageFile` (вдохновлённый паттерном SharedMemory) сериализует данные в JSON и записывает в файлы на диск. Конкурентный доступ управляется через `flock()` с `LOCK_EX` (эксклюзивная блокировка).

LOCK_EX работает корректно

Первый вопрос, который мы проверили: **гарантирует ли `flock()` с `LOCK_EX` действительно взаимное исключение?** Есть ли риск молчаливой записи, перезаписывающей данные?

Ответ ясен: **да, LOCK_EX работает корректно**. Ядро Linux гарантирует, что только один процесс может удерживать `LOCK_EX` на файле в любой момент. Остальные процессы ждут (блокируются), пока блокировка не будет освобождена.

Мы проверили стресс-тестом:

```
// Тест конкуренции flock()
// Запущен с 50 одновременными процессами
$fp = fopen('/tmp/pivot_test.json', 'c+');
if (flock($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);
```

После 10 000 итераций с 50 конкурентными процессами счётчик равнялся ровно 500 000. **Ни одной потерянной записи, ни одного молчаливого перезаписывания.**

Проблема не в корректности. Проблема в **контеншне**.

Узкое место

PmaControl использует файлы-пивоты для хранения состояния контролируемых серверов в реальном времени. Структура выглядит так:

```
/var/lib/pmacontrol/pivot/
  server_status.json      ← статус ВСЕХ серверов
  server_42_metrics.json  ← детальные метрики сервера 42
  server_43_metrics.json
  ...
```

Проблема — файл `server_status.json`. **Каждый демон-воркер**, собрав метрики сервера, обновляет этот центральный файл новым статусом. Операция:

1. Получить `LOCK_EX` на `server_status.json`
2. Прочитать всё содержимое (JSON всех серверов)
3. Изменить запись нужного сервера
4. Перезаписать файл целиком
5. Освободить блокировку

С 10 серверами и интервалом сбора 10 секунд это проходит. С 100 серверами воркеры начинают **взаимно блокировать друг друга** в ожидании блокировки.

Измерение контеншна

Мы инструментировали `StorageFile` для измерения времени ожидания `flock()`:

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

Результаты с разным количеством серверов:

Количество серверов	Среднее время ожидания flock()	P99
10	0.2 мс	1.1 мс
50	4.8 мс	28 мс
100	18 мс	142 мс
200	67 мс	480 мс
500	312 мс	1.8 с

При более 100 серверах P99 превышает 100мс. При 500 серверах некоторые воркеры ждут почти 2 секунды для записи статуса — при интервале сбора 10 секунд. Это 20% бюджета времени, потраченного на ожидание блокировки.

Почему файл растёт

Файл `server_status.json` содержит состояние всех серверов. При 100 серверах он весит примерно 200 КБ. При 500 серверах — примерно 1 МБ.

Каждое обновление:

1. **Читает 1 МБ** JSON
2. **Парсит 1 МБ** в PHP-структуру
3. **Изменяет 2 КБ** (один сервер)
4. **Сериализует 1 МБ** в JSON
5. **Записывает 1 МБ** на диск

Соотношение абсурдно: 2 КБ полезных данных при 4 МБ ввода-вывода.

Решение: шардинг по server_id

Рекомендация — **фрагментировать файл-пивот по server_id**:

```
/var/lib/pmacontrol/pivot/
  status/
    server_42.json    ← 2 КБ, один сервер
    server_43.json
    server_44.json
    ...
```

Каждый воркер блокирует только файл своего сервера. Больше никакого глобального контеншна.

Измеренный эффект

После шардинга:

Количество серверов	Среднее время ожидания flock()	P99
100	0.1 мс	0.5 мс
200	0.1 мс	0.6 мс
500	0.2 мс	0.8 мс

Контеншн практически полностью исчезает. Время ожидания больше не зависит от числа серверов, а зависит от числа воркеров, собирающих данные **одного и того же** сервера (обычно 1).

Компромисс

Дашборд теперь должен читать N файлов вместо одного для отображения общей картины. Код чтения переходит от:

```
// До: один файл
$allStatus = json_decode(file_get_contents('pivot/server_status.json'), true);
```

к:

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

Больше кода, но чтение естественно неблокирующее (нет необходимости в `LOCK_EX` при чтении благодаря атомарным записям через `rename()`).

За пределами файловой системы: Redis и memcached

Для крупных развёртываний (более 500 серверов) файловый подход достигает своих пределов даже с шардингом:

- **Латентность ввода-вывода**: каждая запись затрагивает диск (если не считать page cache Linux)
- **Давление на inode**: 500 файлов-пивотов = 500 inode
- **Нет TTL**: файлы-пивоты удалённых серверов остаются до ручной очистки

Следующий естественный шаг — замена `StorageFile` на бэкенд **Redis** или **memcached**:

```
// Абстрактный интерфейс
interface StorageBackend {
    public function get(string $key): ?array;
```

```
    public function set(string $key, array $data, int $ttl = 0): void;
}

// Файловая реализация (текущая)
class StorageFile implements StorageBackend { ... }

// Redis-реализация (будущая)
class StorageRedis implements StorageBackend { ... }
```

Redis устраняет проблемы контеншна (атомарные операции на стороне сервера), TTL (нативное истечение срока) и производительности (всё в памяти).

Почему нельзя сразу перейти на Redis

PmaControl проектировался для **простой установки**: без внешних зависимостей, один PHP-сервер, без Redis и RabbitMQ. Файловый подход позволяет установку на минимальный Debian без предварительных требований.

Добавление Redis как обязательной зависимости нарушило бы эту философию. Принятое решение — сохранить `StorageFile` как бэкенд по умолчанию (с шардингом) и предложить `StorageRedis` как опцию для крупных развёртываний.

Сводка рекомендаций

Масштаб	Рекомендация	Бэкенд
1-50 серверов	Единый файл-пивот	StorageFile
50-200 серверов	Шардинг по server_id	StorageFile (sharded)
200-500 серверов	Шардинг + быстрый SSD	StorageFile (sharded)
500+ серверов	Redis / memcached	StorageRedis

Заключение

`flock()` с `LOCK_EX` работает корректно — нет молчаливого перезаписывания. Но контеншн на общем файле-пивоте для всех воркеров — реальная проблема при более 100 серверах.

Решение — шардинг по server_id: каждый воркер блокирует свой собственный файл, устраняя глобальный контеншн. Для очень крупных развёртываний Redis берёт на себя эту роль.

Файловая система — неплохой выбор для разделяемой памяти в PHP. Просто нужно знать, когда она достигает своих пределов.