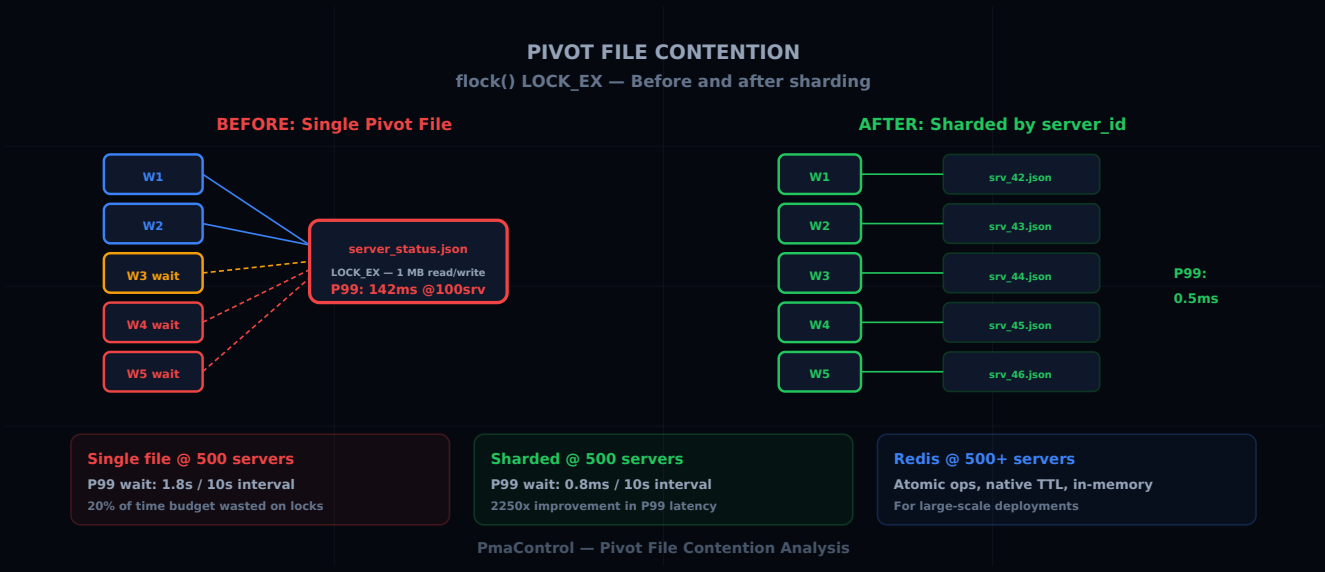
Shared Locks and Pivot Files: Contention in PmaControl

Aurélien LEQUOY · March 19, 2026

PMACONTROL PHP CONCURRENCY PERFORMANCE ARCHITECTURE



The Shared Memory Problem in PHP

PmaControl is a monitoring tool written in PHP. Its daemons collect metrics from tens or hundreds of MariaDB / MySQL instances, then store the results for the web dashboard.

PHP does not have native shared memory between processes (unlike Go with goroutines or Java with threads). Each PHP worker is an independent process. To share data between the collector daemon and the web server, PmaControl uses **pivot files** — a homegrown shared memory implementation via the filesystem.

The `StorageFile` class (inspired by the SharedMemory pattern) serializes data to JSON and writes it to files on disk. Concurrent access is managed by `flock()` with `LOCK_EX` (exclusive lock).

LOCK_EX Works Correctly

First question we verified: **does `flock()` with `LOCK_EX` truly guarantee mutual exclusion?** Is there a risk of silent writes overwriting data?

The answer is clear: **yes, LOCK_EX works correctly**. The Linux kernel guarantees that only one process can hold a `LOCK_EX` on a file at any given time. Other processes wait (block) until the lock is released.

We verified with a stress test:

```
// Concurrency test on flock()
// Launched with 50 simultaneous processes
$fp = fopen('/tmp/pivot_test.json', 'c+');
if (flock($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);
```

After 10,000 iterations with 50 concurrent processes, the counter was exactly 500,000. **No lost writes, no silent overwrites.**

The problem is not correctness. It is **contention**.

The Bottleneck

PmaControl uses pivot files to store the real-time state of monitored servers. The structure looks like:

```
/var/lib/pmacontrol/pivot/
  server_status.json      ← status of ALL servers
  server_42_metrics.json  ← detailed metrics for server 42
  server_43_metrics.json
  ...
```

The problem is the `server_status.json` file. **Each daemon worker**, after collecting metrics from a server, updates this central file with the new status. The operation:

1. Acquire `LOCK_EX` on `server_status.json`
2. Read the full content (JSON of all servers)

3. Modify the entry for the relevant server
4. Rewrite the entire file
5. Release the lock

With 10 servers and a 10-second collection interval, this works fine. With 100 servers, the workers start **blocking each other** waiting for the lock.

Measuring Contention

We instrumented `StorageFile` to measure the wait time on `flock()`:

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

Results with different server counts:

Server Count	Average flock() Wait	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

Beyond 100 servers, the P99 exceeds 100ms. At 500 servers, some workers wait almost 2 seconds to write a status — on a 10-second collection interval. That is 20% of the time budget spent waiting for a lock.

Why the File Grows

The `server_status.json` file contains the state of all servers. At 100 servers, it is about 200 KB. At 500 servers, about 1 MB.

Each update:

1. **Reads 1 MB** of JSON

2. **Parses 1 MB** into a PHP structure
3. **Modifies 2 KB** (a single server)
4. **Serializes 1 MB** to JSON
5. **Writes 1 MB** to disk

The ratio is absurd: 2 KB of useful data for 4 MB of I/O.

Solution: Sharding by server_id

The recommendation is to **shard the pivot file by server_id**:

```
/var/lib/pmacontrol/pivot/
  status/
    server_42.json    ← 2 KB, a single server
    server_43.json
    server_44.json
    ...
```

Each worker only needs to lock the file for its server. No more global contention.

Measured Impact

After sharding:

Server Count	Average flock() Wait	P99
100	0.1 ms	0.5 ms
200	0.1 ms	0.6 ms
500	0.2 ms	0.8 ms

Contention virtually disappears. Wait time no longer depends on the number of servers but on the number of workers collecting the **same** server (typically 1).

Trade-off

The dashboard must now read N files instead of one to display the overview. The reading code changes from:

```
// Before: a single file
$allStatus = json_decode(file_get_contents('pivot/server_status.json'), true);
```

to:

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

This is more code, but reading is naturally non-blocking (no `LOCK_EX` needed for reads thanks to atomic writes via `rename()`).

Beyond the Filesystem: Redis and memcached

For large deployments (more than 500 servers), the filesystem approach reaches its limits even with sharding:

- **I/O latency**: each write touches the disk (except for Linux page cache)
- **Inode pressure**: 500 pivot files = 500 inodes
- **No TTL**: pivot files for deleted servers remain until manual cleanup

The natural next step is to replace `StorageFile` with a **Redis** or **memcached** backend:

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

// File implementation (current)
class StorageFile implements StorageBackend { ... }

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

Redis eliminates contention issues (server-side atomic operations), TTL problems (native expiration), and performance concerns (everything in memory).

Why Not Switch to Redis Immediately

PmaControl was designed to be **simple to install**: no external dependencies, a single PHP server, no Redis or RabbitMQ. The pivot file approach allows installation on a minimal Debian without prerequisites.

Adding Redis as a mandatory dependency would break this philosophy. The chosen solution is to keep `StorageFile` as the default backend (with sharding) and offer `StorageRedis` as an option for large deployments.

Recommendation Summary

Scale	Recommendation	Backend
1-50 servers	Single pivot file	StorageFile
50-200 servers	Sharding by server_id	StorageFile (sharded)
200-500 servers	Sharding + fast SSD	StorageFile (sharded)
500+ servers	Redis / memcached	StorageRedis

Conclusion

`flock()` with `LOCK_EX` works correctly — no silent overwrites. But contention on a pivot file shared by all workers is a real problem beyond 100 servers.

The solution is sharding by server_id: each worker locks its own file, eliminating global contention. For very large deployments, Redis takes over.

The filesystem is not a bad choice for shared memory in PHP. You just need to know when it reaches its limits.