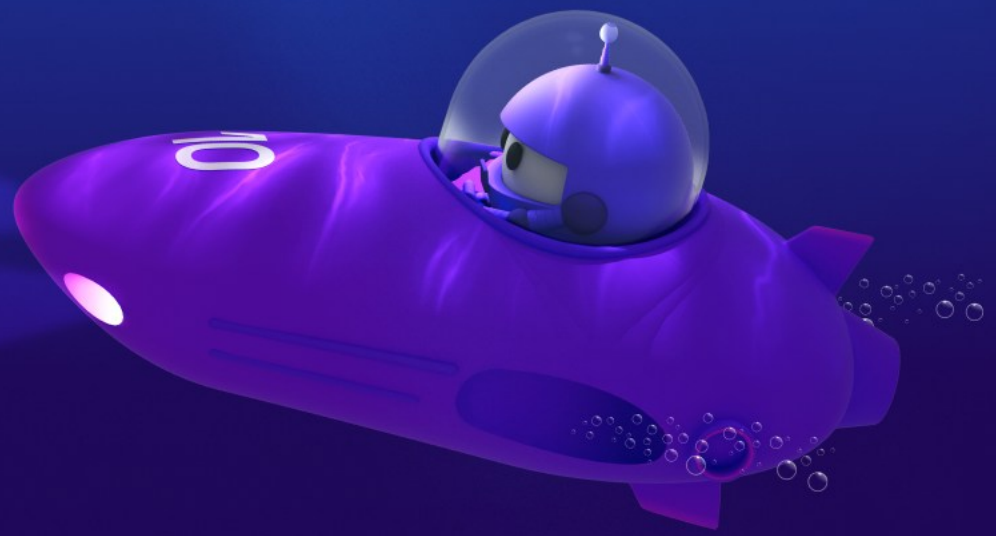
Hybrid Caching in .NET

Jody Donetti



DotNetConf Liguria 2025

Jody Donetti

Code + R&D

Faccio cose (principalmente) sul web da circa 30 anni.

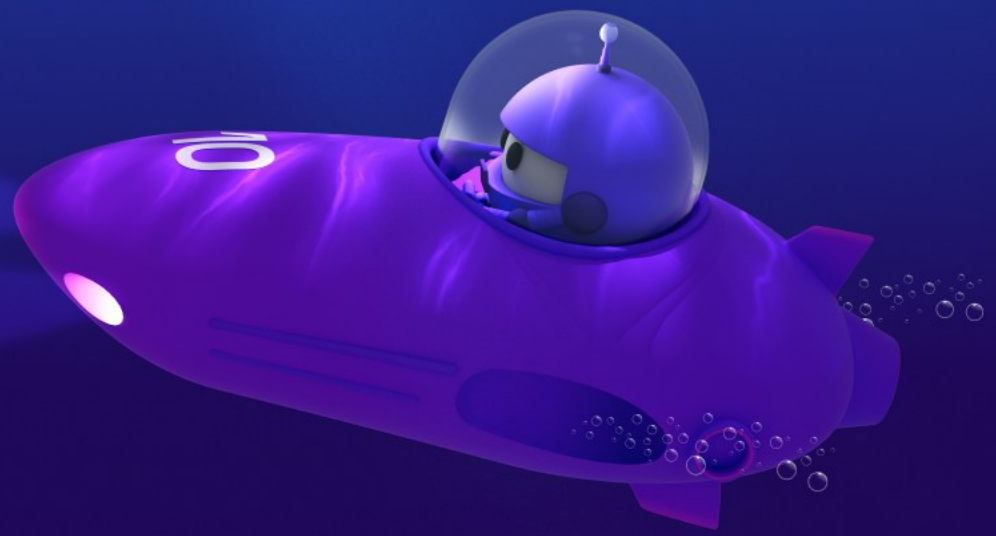
Ho avuto a che fare con la maggior parte dei tipi di cache: memory, distributed, hybrid, HTTP, offline e CDN.

Ho creato FusionCache, una hybrid cache .NET free + OSS.

🏆 Google OSS Award

🏆 Microsoft MVP Award

Caching: Mini Intro



💀 Caching: Mini Intro

Dunque, ridotto all’osso:

Cosa si intende per caching?

Il caching è una **pratica**.

💀 Caching: Mini Intro

I dati di cui abbiamo bisogno sono in una **fonte dati**, tipicamente un **database**.

Normalmente, quando abbiamo bisogno di un dato:

- **ogni richiesta:** andiamo alla fonte (database)

Tutto molto lineare.

Ma un database è tipicamente più **lento** di una **cache**.

 Caching: Mini Intro

Quando facciamo caching:

- **prima richiesta (più lavoro):** andiamo alla fonte (database, più lento) e salviamo in cache
- **richieste successive (meno lavoro):** leggiamo dalla cache (più veloce)

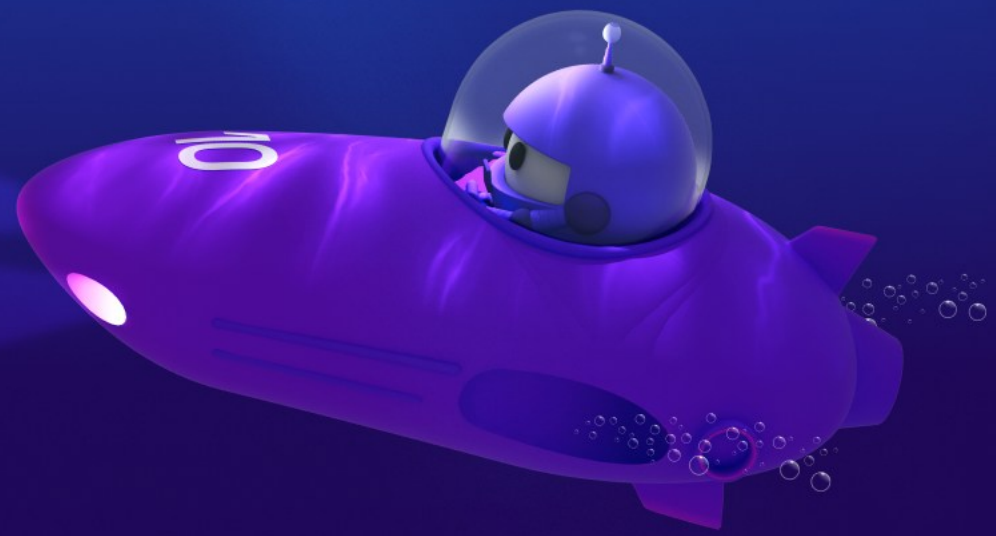
Non dobbiamo usare caching **sempre** o per **tutti** i dati.

Dobbiamo usarlo solo per **alcuni** dati, quelli per cui ha senso (e.g.: scenari **read-heavy**).

Il segreto è trovare il giusto **equilibrio**.

Ok, fine.

Cache Stampede



Cache Stampede

Immaginiamo questo scenario.

Arrivano richieste alla nostra app/servizio, tutte per gli stessi **dati** (non ancora nella cache) e tutte **contemporaneamente**.

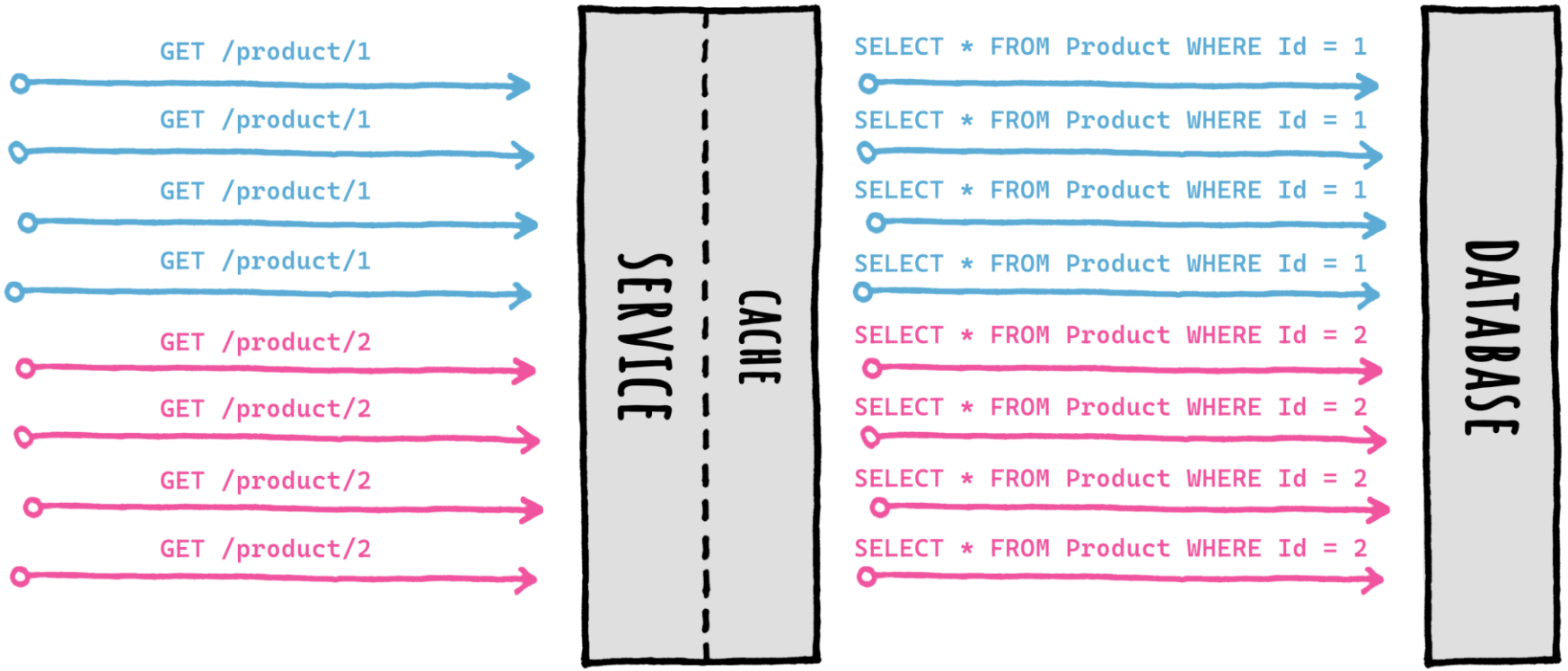
Senza alcuna cura particolare, **ogni richiesta** farebbe:

- **GET:** lettura dalla cache
- **CHECK:** cache hit/miss
- **FACTORY:** database query
- **SET:** scrittura nella cache

Bene ma non benissimo.

Cache Stampede

Fondamentalmente, in caso di **cache miss**, abbiamo questo:



Cache Stampede

Immaginiamo con **100**, **1000** o anche più richieste concorrenti.

Un grande **spreco** di tempo, risorse e un rischio di **overload** per il nostro database.

E magari durante le ore di punta, in un Black Friday.
Perchè ovviamente, no?

Piacere di conoscerti **Cache Stampede**.

Quindi, cosa possiamo fare?

Cache Stampede

Alcune librerie di cache, ma **non tutte**, hanno una **protezione integrata** per la Cache Stampede.

Lo fanno coordinando:

- operazioni sulla cache (get/set)
- esecuzione della factory (query database)

per **la stessa cache key** e nello **stesso momento**, tutto automaticamente

Ma dobbiamo **dare loro** la **possibilità** di proteggerci.

E come?

Cache Stampede

Non facendo **chiamate separate**, che la cache non potrebbe coordinare:

```
// CACHE READ
var product = cache.Get<Product>($"product:{id}");

// CACHE MISS CHECK
if (product is null)
{
    // DATABASE READ
    product = GetProductFromDb(id);
    // CACHE WRITE
    cache.Set<Product>($"product:{id}", product);
}
```

🛡️ Cache Stampede

Ma **facendo** invece una **chiamata singola** passando una **factory**:

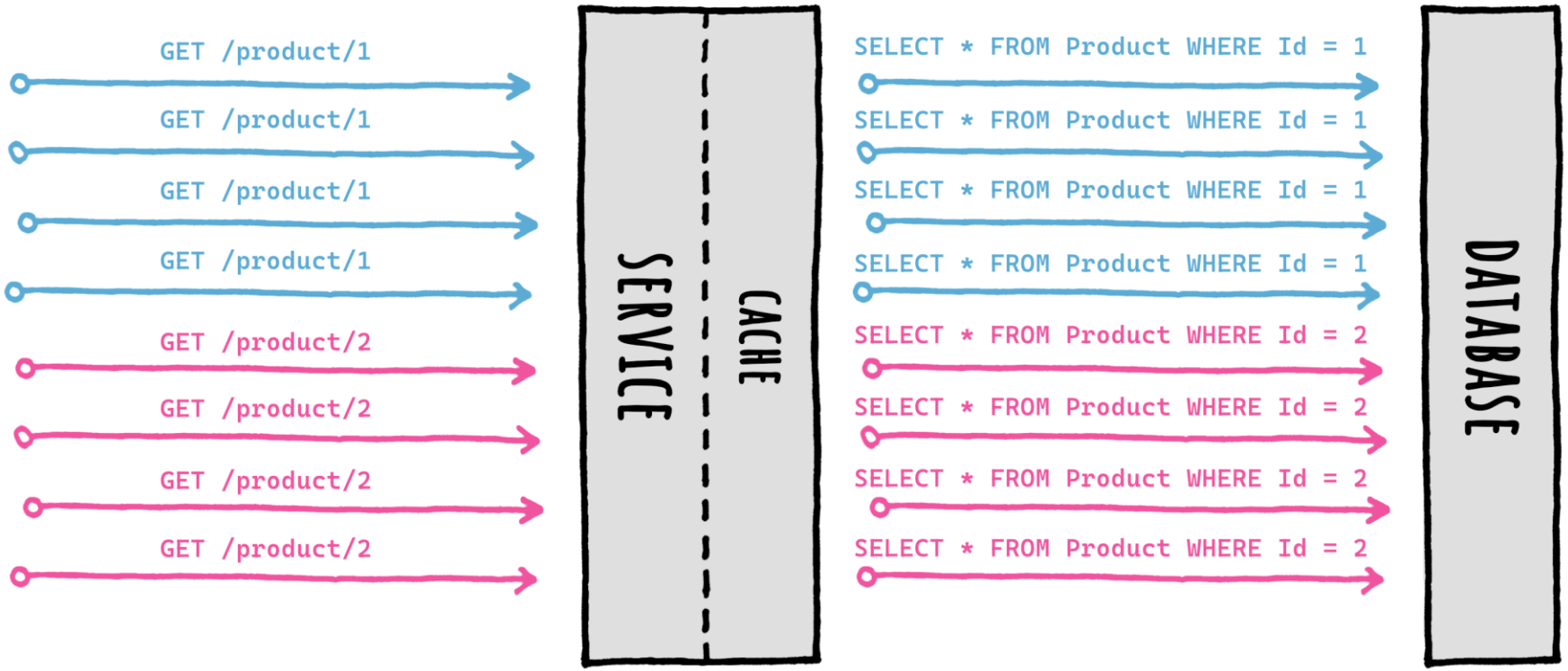
```
var product = cache.GetOrSet<Product>(
    $"product:{id}",
    // FACTORY (DATABASE READ)
    _ => GetProductFromDb(id)
);
```

Questo permette alla cache di **coordinare** il tutto.

ℹ️ **NOTA:** il metodo può chiamarsi anche `GetOrCreate()`, `GetOrAdd()`, etc

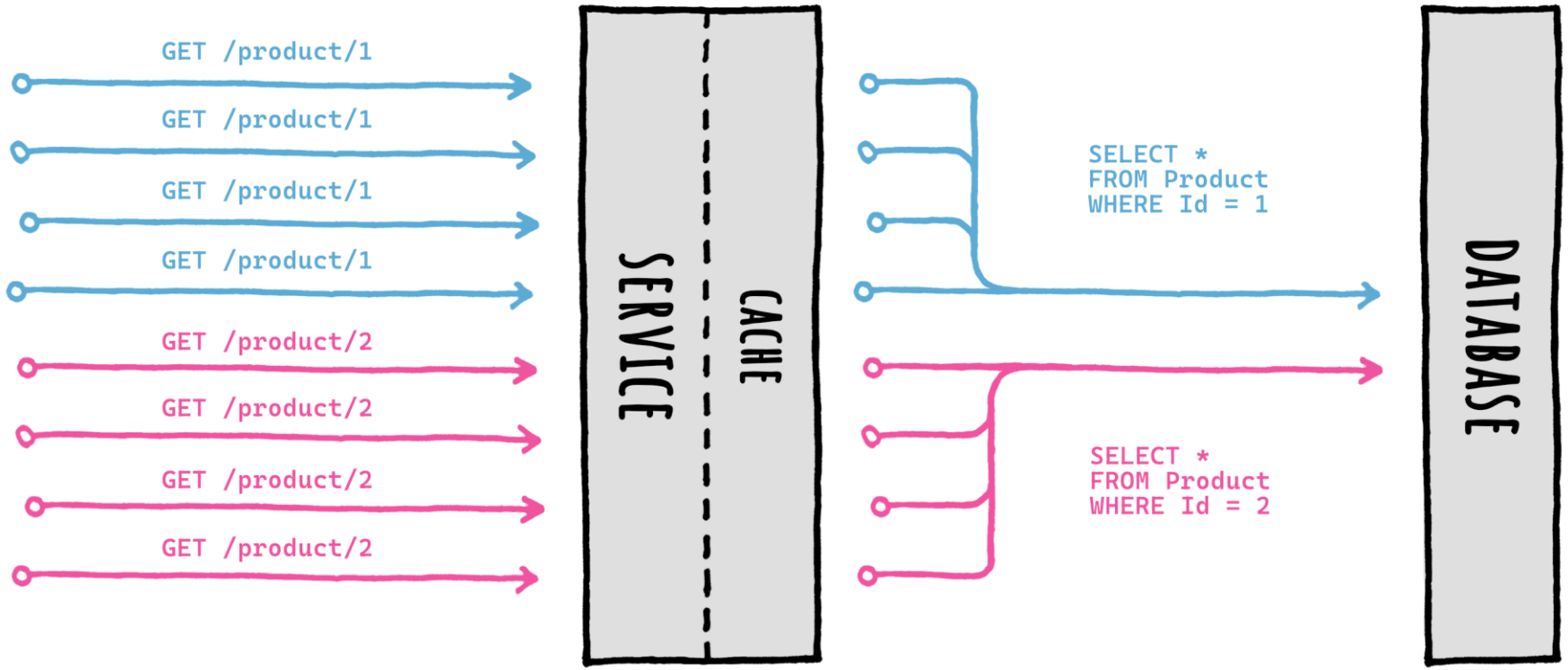
Cache Stampede

In pratica passando da questo:



Cache Stampede

A questo:



Cache Stampede

È una forma di **request coalescing**, ossia: più richieste (logiche) si fondono in una sola (fisica).

Attenzione però, spesso si pensa che:

- **SE** la libreria fornisce un metodo `GetOrSet(key, factory)` o simile
- **ALLORA** la libreria protegge da cache stampede

Questo è **falso**.

🛡️ Cache Stampede

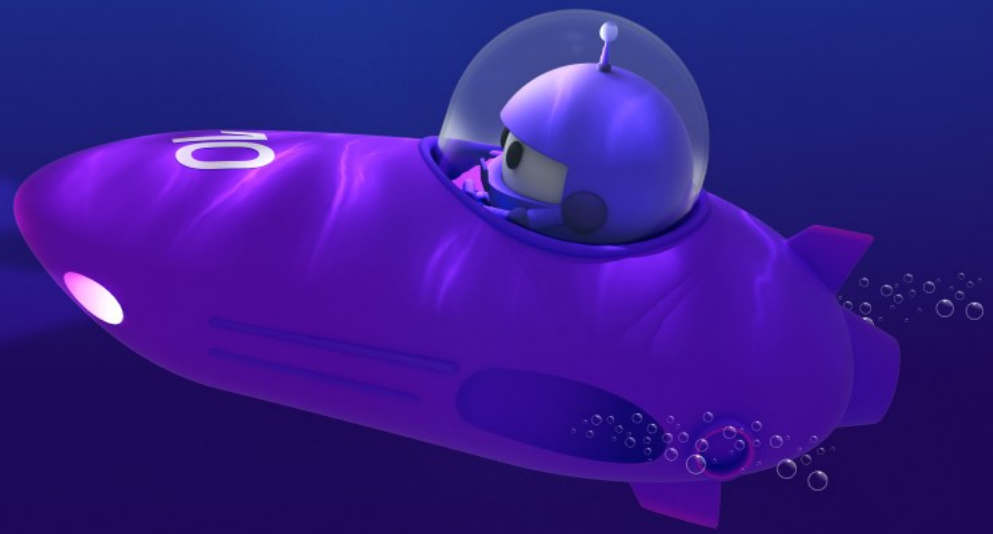
Per esempio:

- ⛔ **MemoryCache:** nessuna protezione, nemmeno con `GetOrCreate()`/`GetOrCreateAsync()`
- ✅ **FusionCache:** protezione con `GetOrSet()`/`GetOrSetAsync()`
- ✅ **HybridCache:** protezione con `GetOrCreateAsync()`

Ricordiamoci di controllare la libreria di cache che usiamo.

Ok, ma quindi hybrid caching?

Hybrid cosa?



🤔 Hybrid cosa?

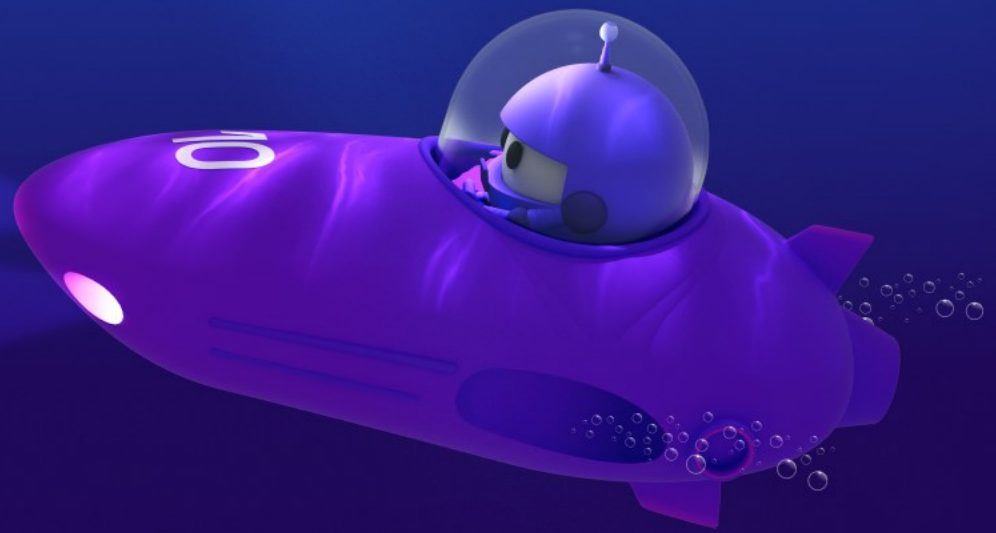
Cos’è una "cache ibrida"?

E ha senso usarne una?

Per capirlo conosciamo i 3 tipi principali di cache:

- memory cache
- distributed cache
- hybrid/multi-level cache

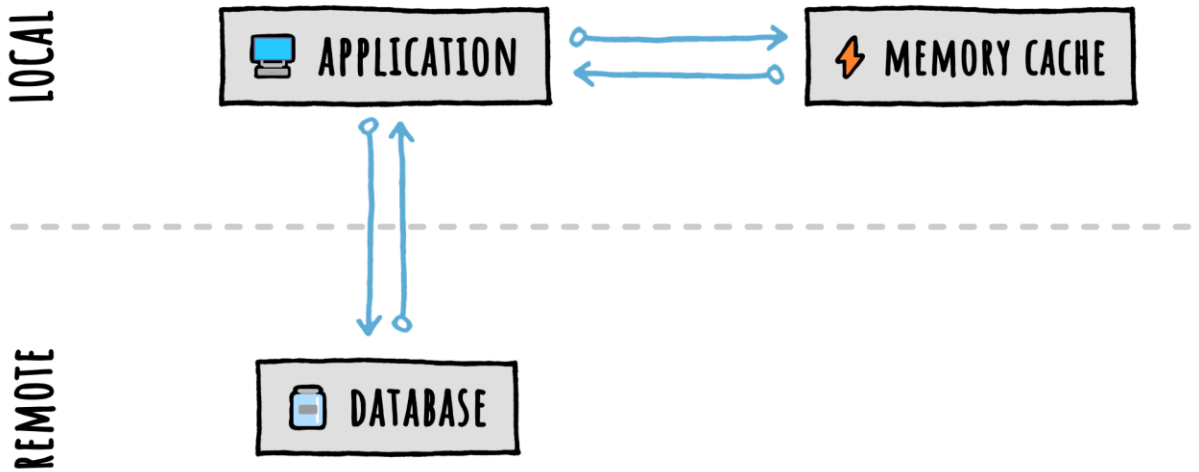
Le Memory Cache



Le Memory Cache

Le **memory cache** memorizzano i dati in memoria.
E non solo "in memoria", ma nella **stessa** memoria dell'applicazione che lo utilizza.

Pensiamole come un `Dictionary<K,V>` più qualche forma di **eviction**.



⚡ Le Memory Cache

Possiamo usarle così:

```
var product = cache.GetOrCreate<Product>(
    $"product:{id}",
    _ => GetProductFromDb(id),
    options
);
```

⚡ Le Memory Cache

Alcuni esempi di **memory cache** in .NET:

📦 **BitFaster.Caching**
github.com/bitfaster/BitFaster.Caching

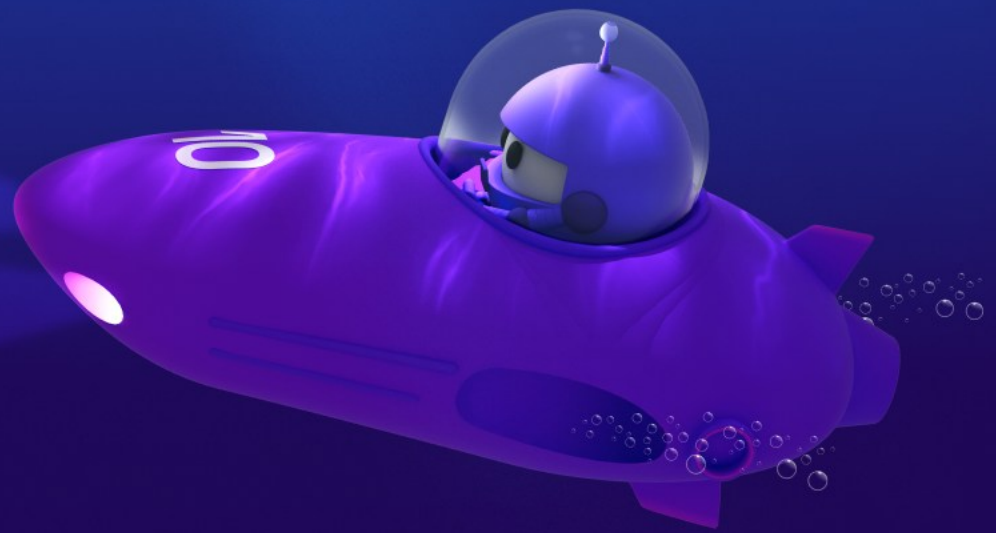
📦 **FastCache**
github.com/jitbit/FastCache

📦 **fast-cache**
github.com/neon-sunset/fast-cache

📦 **LazyCache**
github.com/alastairtree/LazyCache

Ⓜ️ **Microsoft MemoryCache**

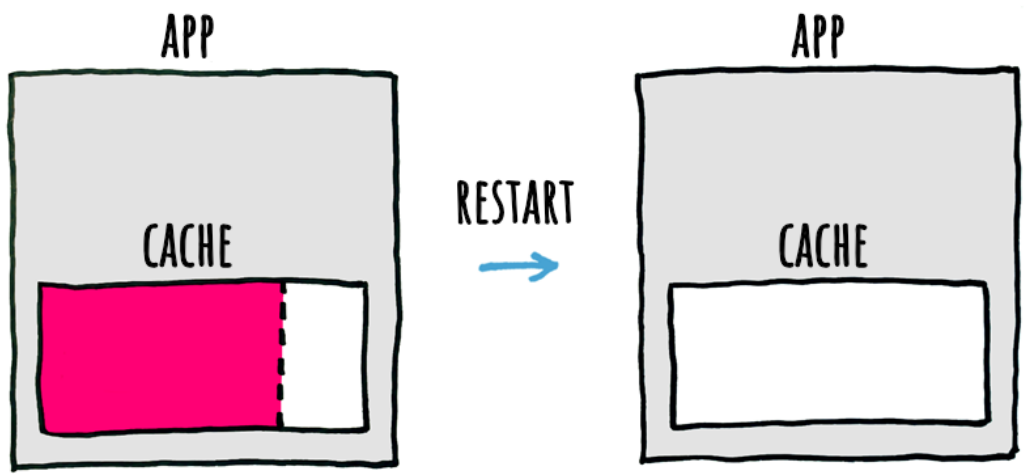
Cold Start



🥶 Cold Start

Ok, usiamo una **memory cache**.
Cosa succede quando la nostra **app si riavvia**?

Questo:



La cache in memoria torna ad essere **vuota**.

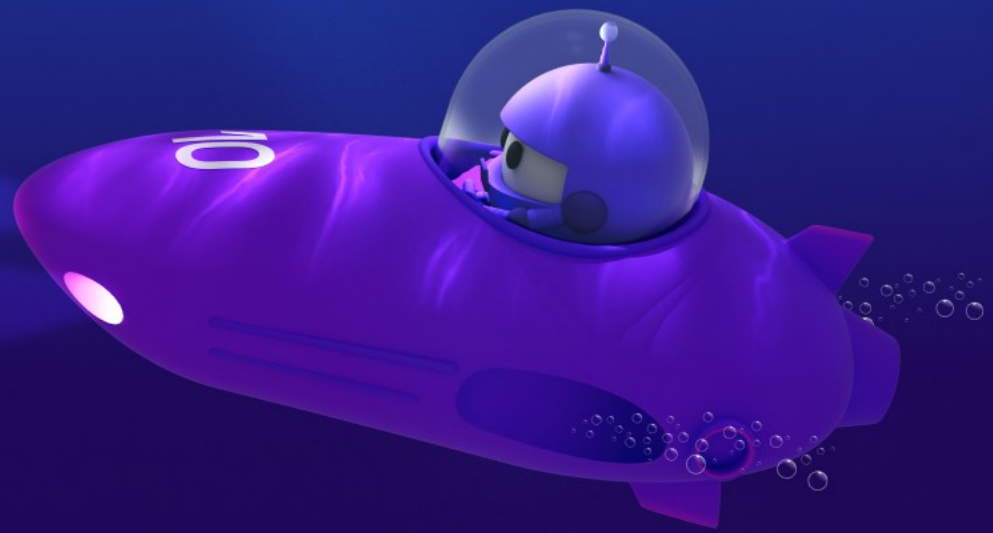
🥶 Cold Start

Una **memory cache** è fondamentalmente un `Dictionary<K,V>`, e quindi va **ripopolata** ad ogni riavvio dell'applicazione.

Questo significa **più query** verso il **database**.

Ok, altro?

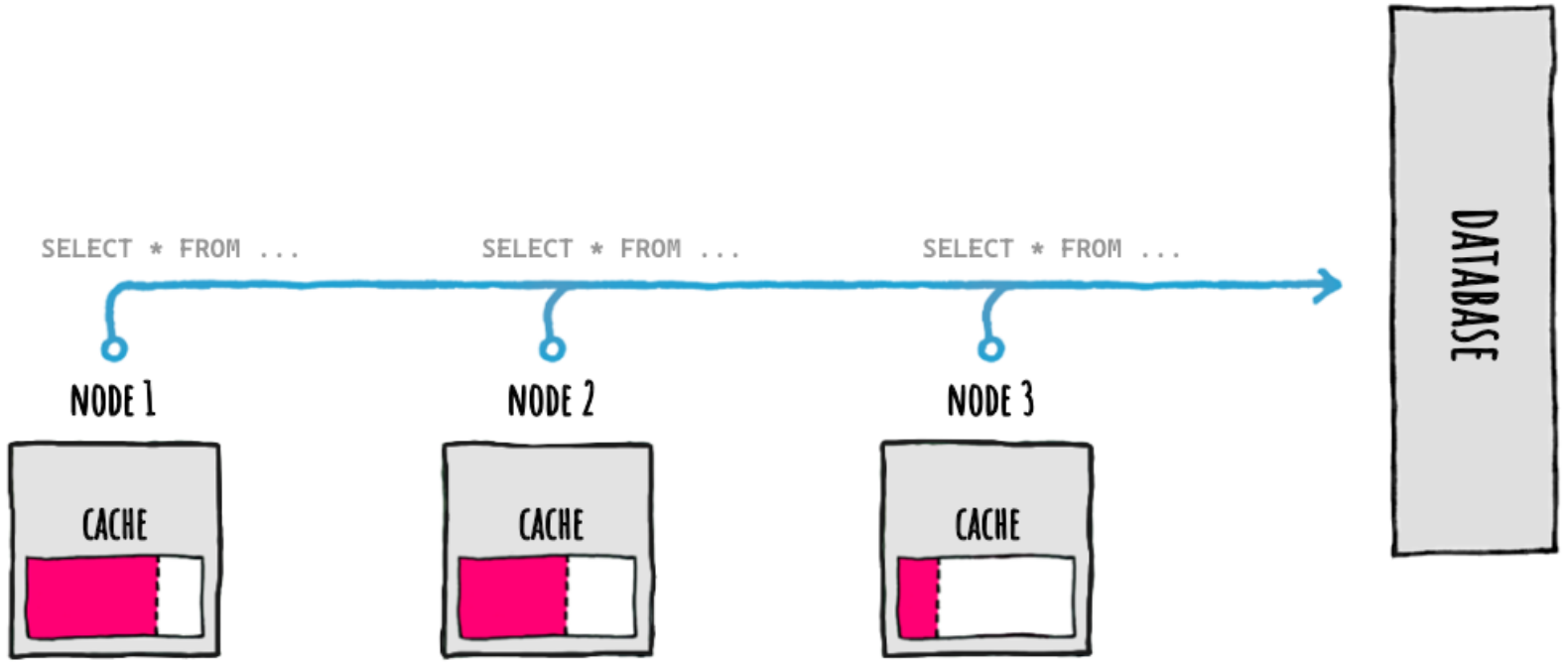
Scalabilità Orizzontale



Scalabilità Orizzontale

Cosa succede se la nostra app è distribuita su più **istanze/nodi/pod**?

Questo:



Ogni istanza/nodo/pod ha **la propria** cache locale.

🪜 Scalabilità Orizzontale

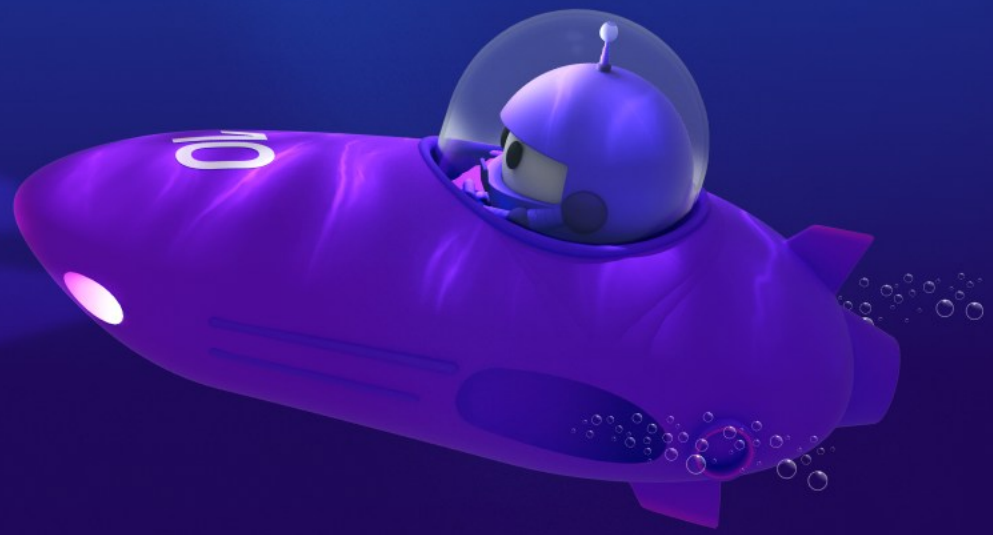
Ogni istanza viene **popolata** prendendo i dati dal **database**.

Questo perchè i dati nella cache **non sono condivisi**.

E questo, di nuovo, significa **più query** verso il **database**.

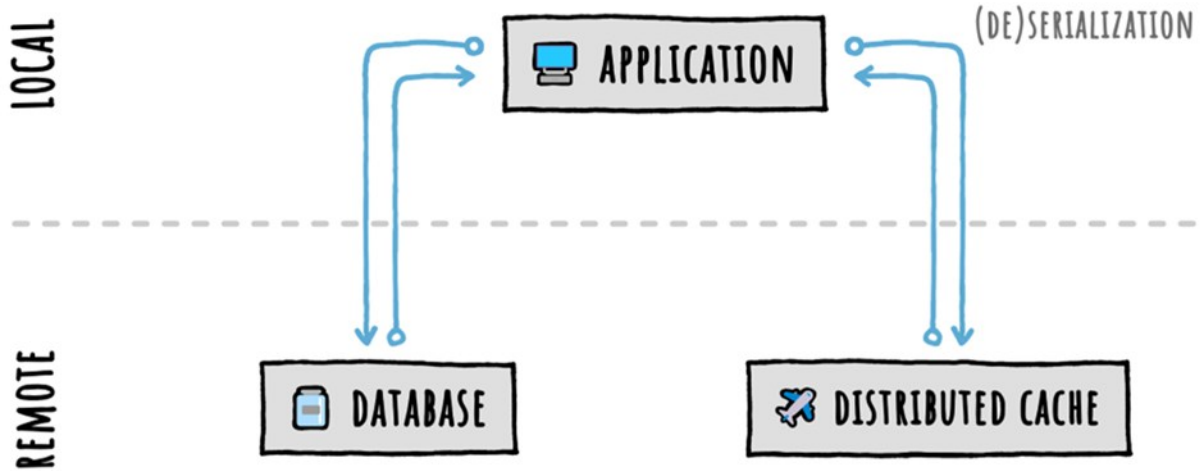
Ok, cosa possiamo fare?

Le Distributed Cache



✈️ Le Distributed Cache

Le **cache distribuite** rappresentano **key-value store remoti** (Redis, Memcached).
Come un database ma **più semplici**, con meno feature: per questo molto **più performanti**:



✈️ Le Distributed Cache

In .NET è principalmente `IDistributedCache` e relative implementazioni:

```
public interface IDistributedCache
{
  byte[]? Get(string key);
  Task<byte[]?> GetAsync(string key, CancellationToken token = default);

  void Set(string key, byte[] value, DistributedCacheEntryOptions options);
  Task SetAsync(string key, byte[] value, DistributedCacheEntryOptions options, CancellationToken token = default);

  void Refresh(string key);
  Task RefreshAsync(string key, CancellationToken token = default);

  void Remove(string key);
  Task RemoveAsync(string key, CancellationToken token = default);
}
```

Parlano in binario, tramite `byte[]`.

 Le Distributed Cache

In .NET è principalmente `IDistributedCache` e relative implementazioni:

```
public interface IDistributedCache
{
  byte[]? Get(string key);
  Task<byte[]?> GetAsync(string key, CancellationToken token = default);

  void Set(string key, byte[] value, DistributedCacheEntryOptions options);
  Task SetAsync(string key, byte[] value, DistributedCacheEntryOptions options, CancellationToken token = default);

  void Refresh(string key);
  Task RefreshAsync(string key, CancellationToken token = default);

  void Remove(string key);
  Task RemoveAsync(string key, CancellationToken token = default);
}
```

Parlano in binario, tramite `byte[]`.

✈️ Le Distributed Cache

Possiamo usarle così:

```
Product product;
var payload = distributedCache.Get($"product:{id}");
if (payload is not null)
{
    product = JsonSerializer.Deserialize<Product>(payload);
}
else
{
    // WARNING: NO STAMPEDE PROTECTION
    product = GetProductFromDb(id);
    payload = JsonSerializer.SerializeToUtf8Bytes<T?>(product);
    distributedCache.Set($"product:{id}", payload);
}
```

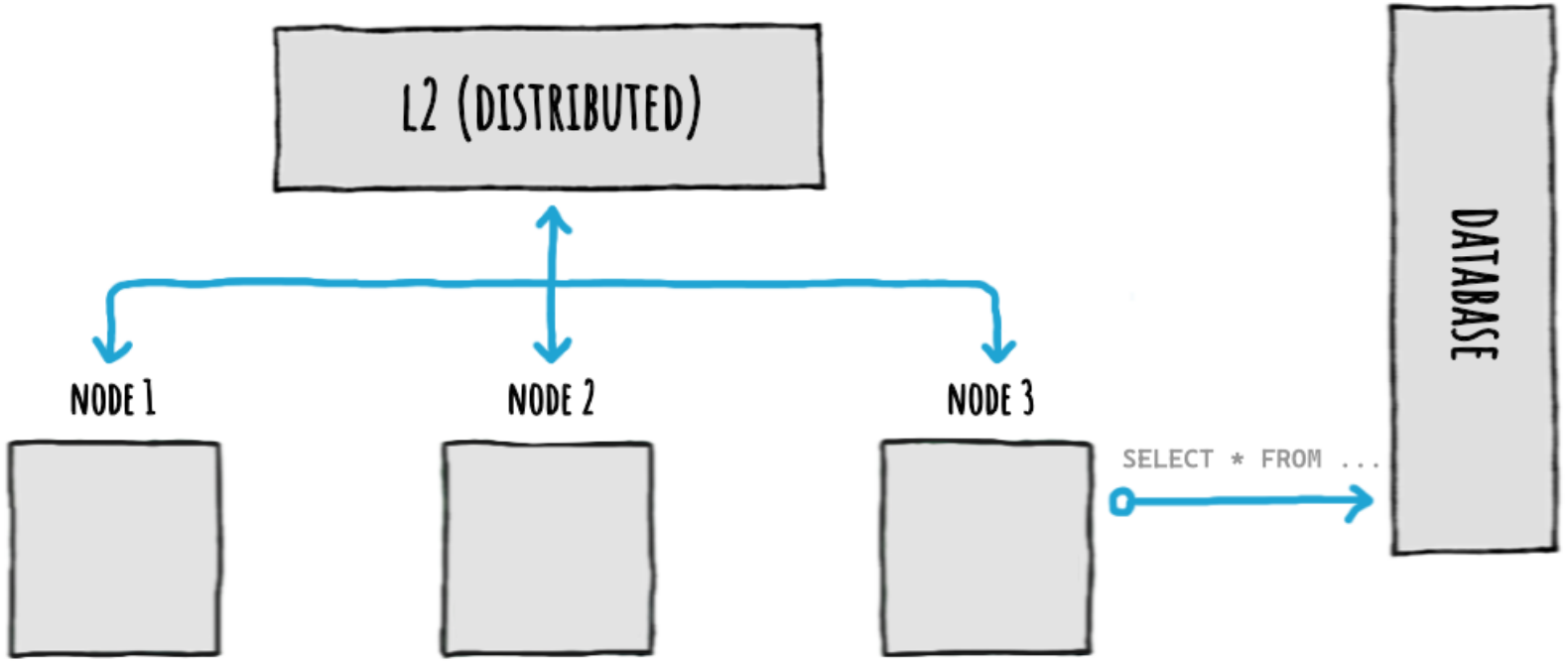
Quindi si: è necessario **più codice**.

✈️ Le Distributed Cache

Poiché le distributed cache sono **remote**, i dati vivono **al di fuori** della memoria dell'app.

Quindi:

- i **cold start** non svuotano **la cache**
- i dati vengono **condivisi** fra **diversi nodi**



✈️ Le Distributed Cache

Alcuni esempi di **distribuited cache** in .NET (implementazioni di `IDistributedCache`):

📦 **EnyimMemcachedCore** (per Memcached)
github.com/cnblogs/EnyimMemcachedCore

📦 **MongoDbCache** (per MongoDB)

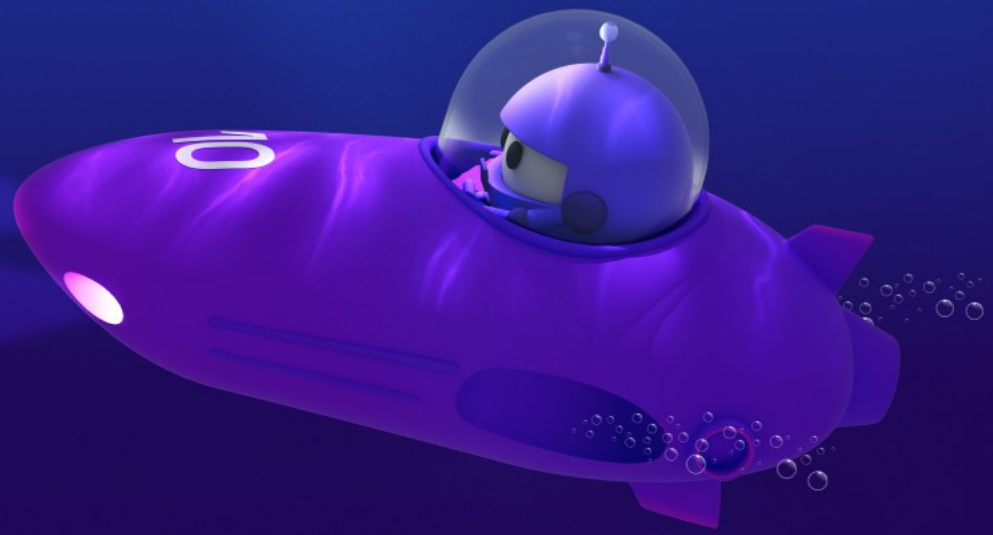
github.com/outmatic/MongoDbCache

📦 **NeoSmart.Caching.Sqlite** (per SQLite, interessante!)
github.com/neosmart/SqliteCache

📦 **AWS. AspNetCore.DistributedCacheProvider** (per Amazon DynamoDB)
github.com/aws/aws-dotnet-distributed-cache-provider/

Ⓜ️ **Microsoft.Extensions.Caching.StackExchangeRedis** (per Redis)

Usiamo Solo Distributed Cache?



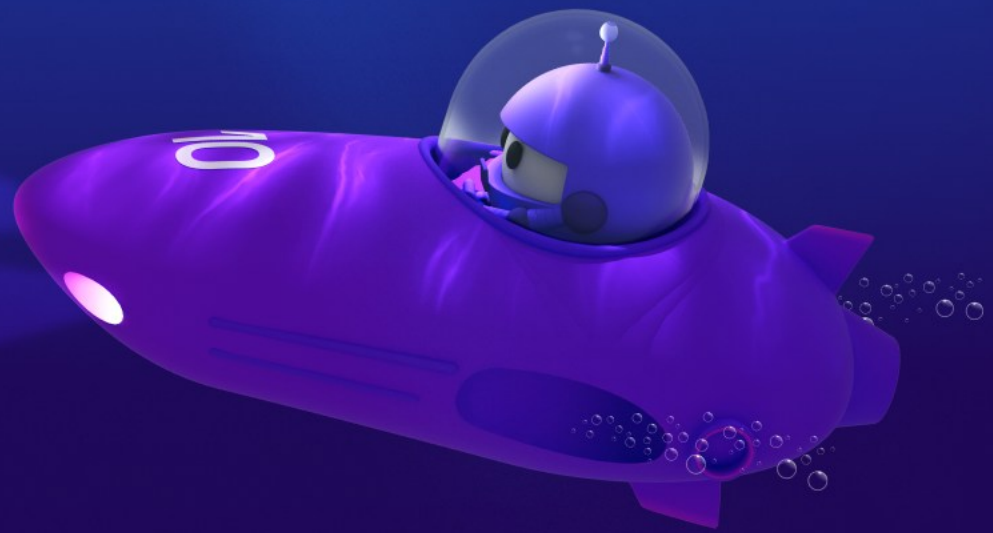
Usiamo Solo Distributed Cache?

Usando **direttamente** una distributed cache dobbiamo considerare:

- **codice:** più codice da scrivere/mantenere
- **performance:** per ogni chiamata abbiamo network + serializzazione
- **availability:** non sempre disponibile (Fallacies Of Distributed Computing)
- **cache stampede:** nessuna protezione

E qui è dove entrano in gioco le **hybrid cache**.

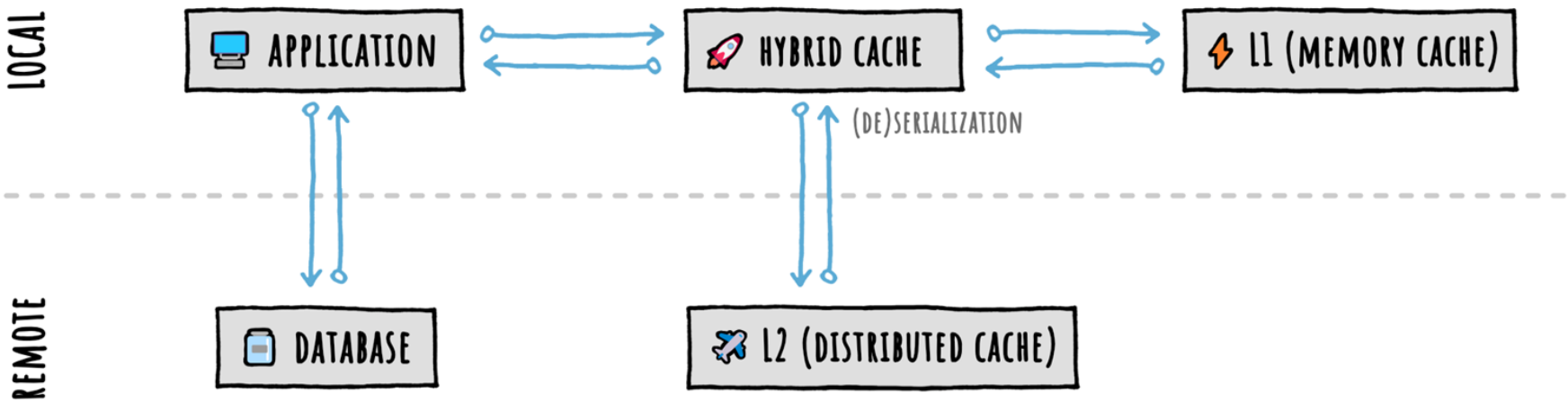
Le Hybrid Cache



🚀 Le Hybrid Cache

Le **hybrid cache** sono le cache più avanzate.
Combinano insieme il meglio di entrambi i mondi: memory (L1) + distributed (L2).

La "danza" tra i due livelli è gestita automaticamente.



🚀 Le Hybrid Cache

Possiamo usarle così:

```
var product = cache.GetOrSet<Product>(
    $"product:{id}",
    _ => GetProductFromDb(id),
    options
);
```

🚀 Le Hybrid Cache

Alcuni esempi di cache **ibride/multi-livello** in .NET:

📦 **CacheTower (multi-level)**
github.com/TurnerSoftware/CacheTower

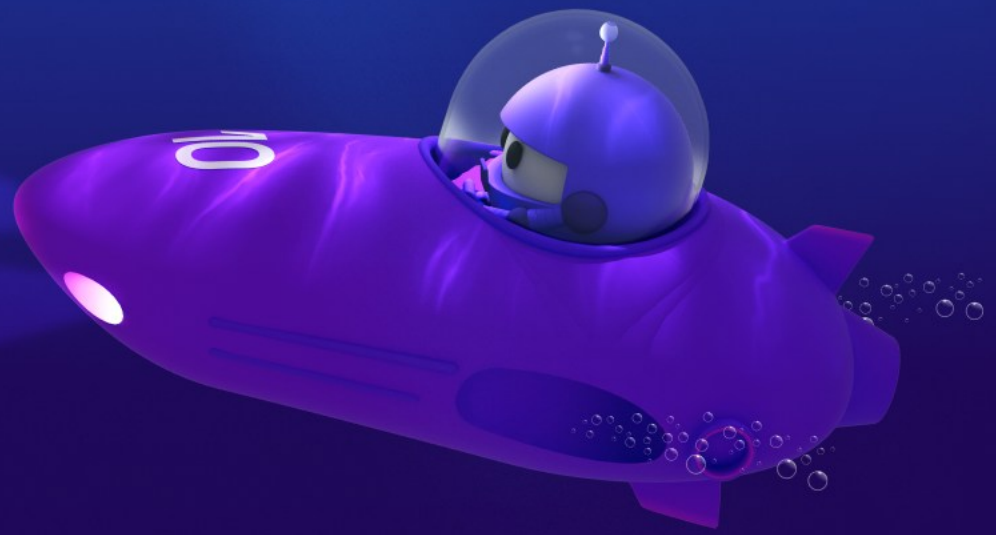
📦 **CacheManager (multi-level)**
github.com/MichaCo/CacheManager

📦 **EasyCaching (multi-level)**
github.com/dotnetcore/EasyCaching

📦 **FusionCache (hybrid)**
github.com/ZiggyCreatures/FusionCache

Ⓜ️ **Microsoft HybridCache (hybrid)**

Hybrid VS Multi-Level



🚀 Hybrid VS Multi-Level

Chiariamo un momento: sono **simili,** ma **diverse**.

In generale:

- **multi-level:** qualsiasi numero di livelli, ognuno di qualsiasi tipo
- **hybrid:** L1 (memory) o L1+L2 (memory+distributed)

🚀 Hybrid VS Multi-Level

Come sviluppatori, possiamo pensarle così (pseudo-codice):

```
class MultiLevelCache
{
    List<ICacheLevel> Levels { get; }
}
```

```
class HybridCache
{
    IMemoryCache L1 { get; }
    IDistributedCache? L2 { get; }
}
```

🚀 Hybrid VS Multi-Level

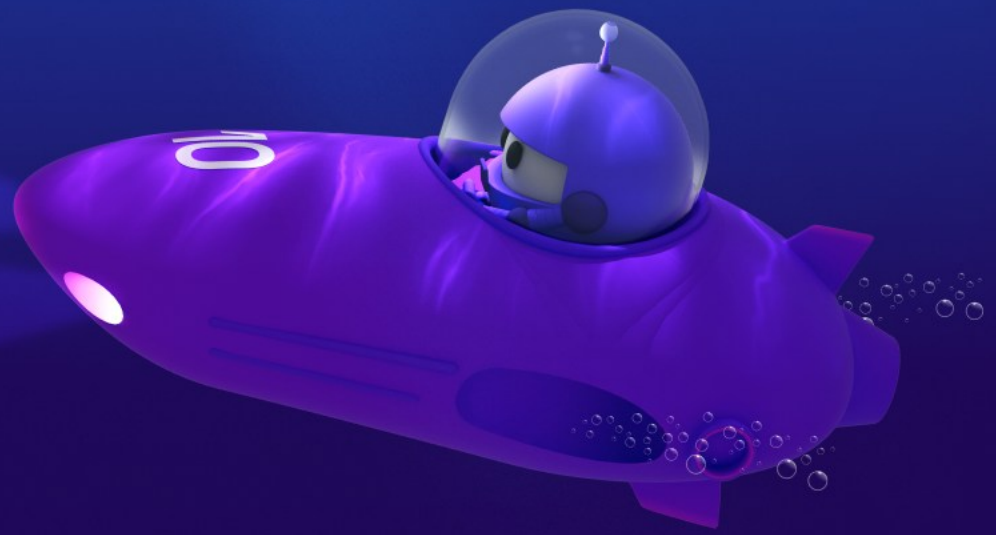
Le cache ibride possono sembrare più «limitate», ma sono opinionate e, in effetti, più potenti.

Ecco perché:

- le limitazioni sono **pragmatiche**, senza impatti reali
- queste garantiscono **basi più solide** su cui costruire
- consentono un **design più ricco** con **feature** più **avanzate**
- offrendo comunque un controllo più **granulare** (x es: skip L1/L2 per-call)
- in generale funzionano, anche in scenari complessi

Tutto sommato, sono (imho) il giusto **equilibrio**.

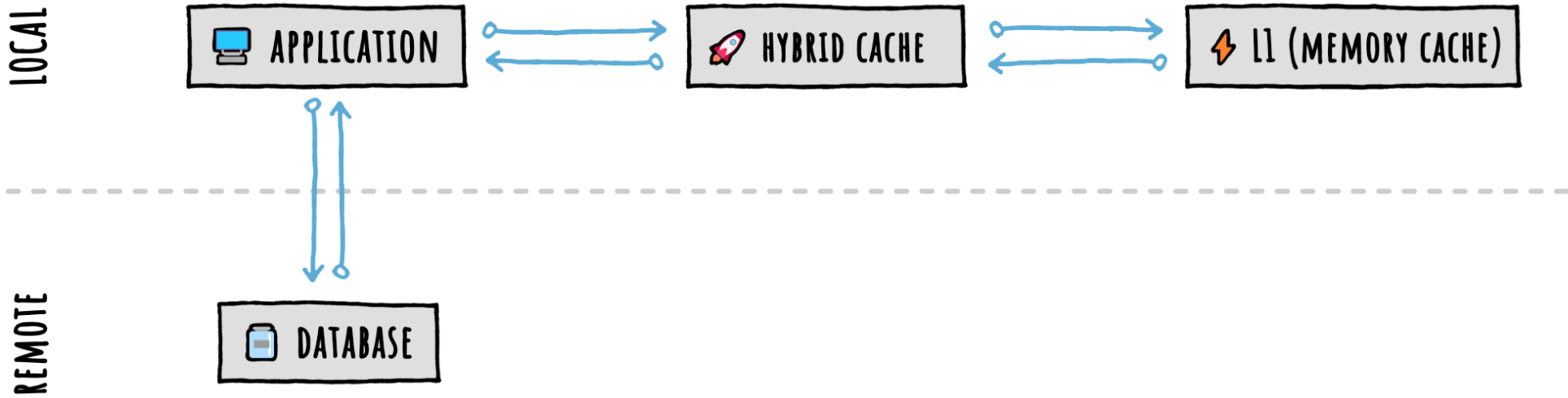
Hybrid != L1+L2



🚀 Hybrid != L1+L2

Usando una hybrid cache non siamo **costretti** a usare più livelli (L1+L2).

Possiamo anche usare **solo L1**:



Ok, ma... perché?

🚀 Hybrid != L1+L2

Le cache ibride sono più **high level**.

Quindi, anche se **dipende** dalla libreria specifica, in generale possiamo **aspettarci**:

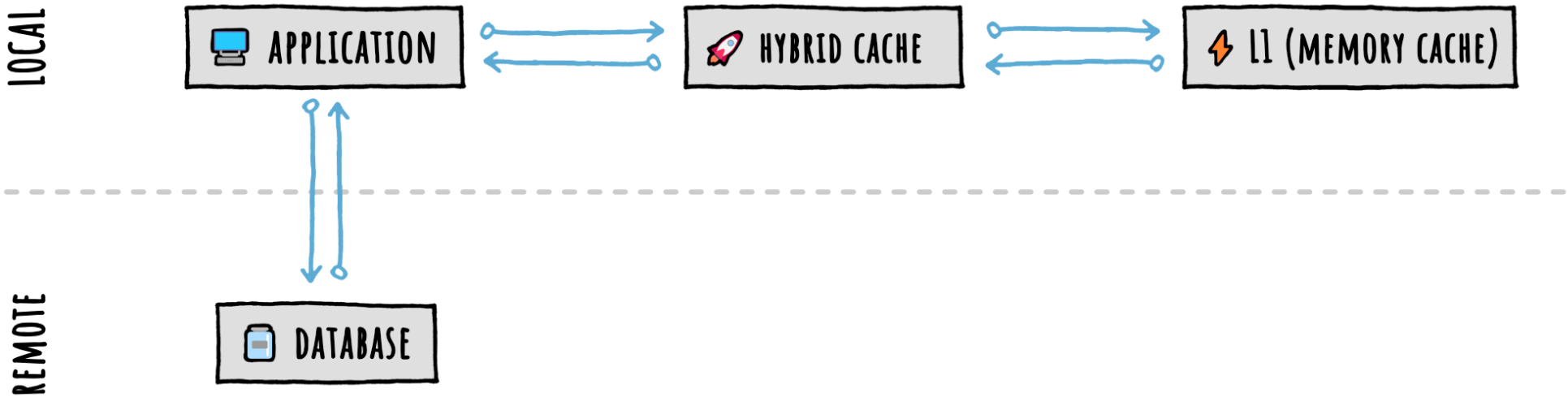
- più feature
- feature più avanzate
- observability
- etc

Ma soprattutto possiamo passare in modo **transparente** da uno a più livelli.

Il tutto **senza cambiare** il **nostro codice**.

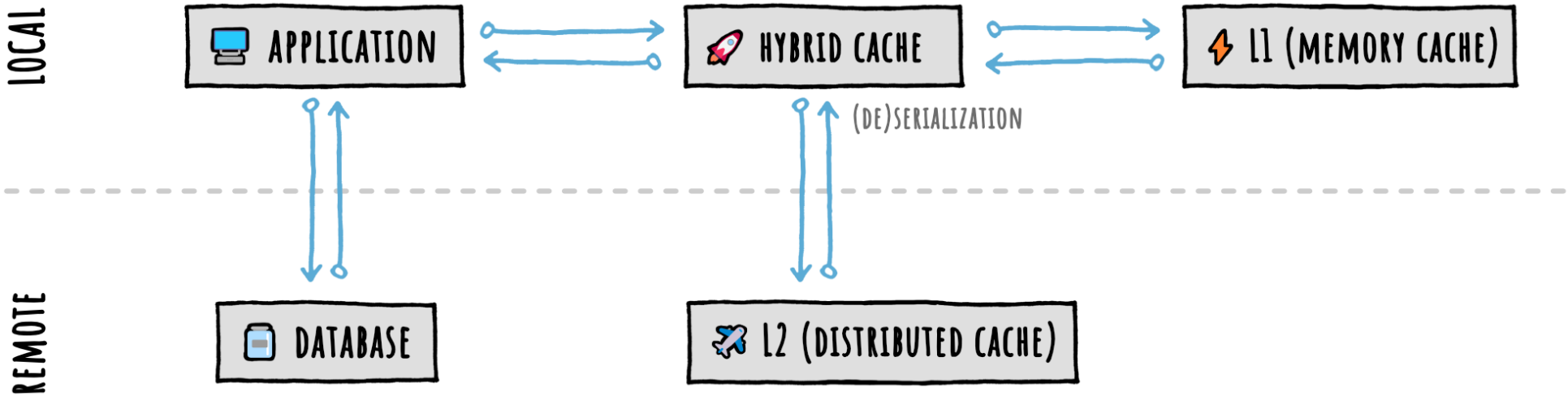
🚀 Hybrid != L1+L2

Ovvero: non importa se abbiamo un setup con **solo L1**...



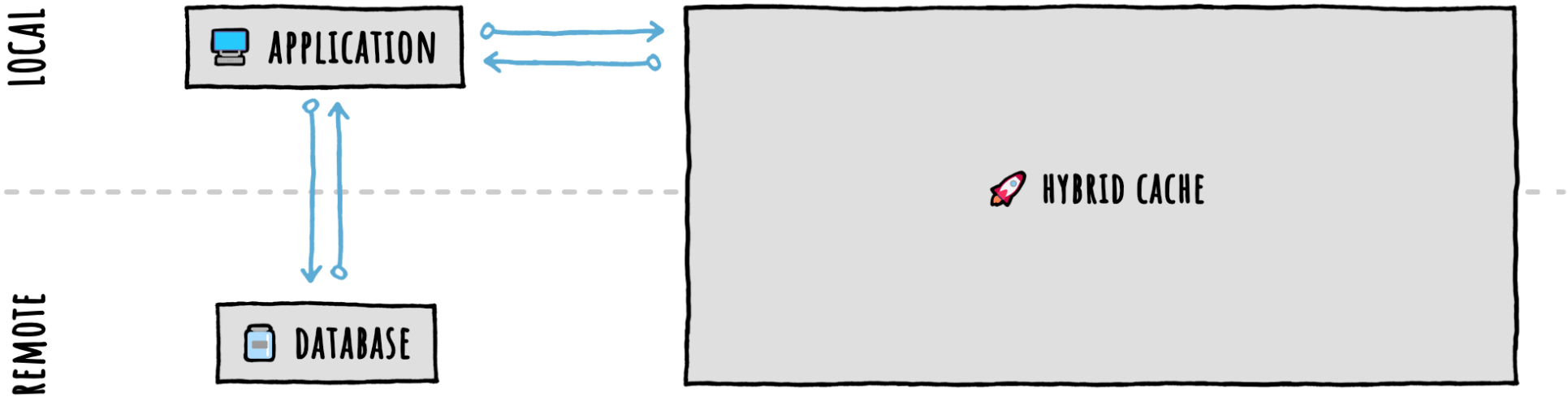
🚀 Hybrid != L1+L2

... o **L1+L2** perché, in entrambi i casi...



🚀 Hybrid != L1+L2

... possiamo semplicemente usare una singola **API unificata**:



🚀 Hybrid != L1+L2

Fondamentalmente, per lavorare con **solo L1**:

```
var product = cache.GetOrSet<Product>(
    $"product:{id}",
    _ => GetProductFromDb(id),
    options
);
```

🚀 Hybrid != L1+L2

Mentre con **L1+L2**, di cui L2 su **Redis** e serializzazione **Protobuf**:

```
var product = cache.GetOrSet<Product>(
    $"product:{id}",
    _ => GetProductFromDb(id),
    options
);
```

🚀 Hybrid != L1+L2

E con **L1+L2**, di cui L2 su **Memcached** e serializzazione **JSON**:

```
var product = cache.GetOrSet<Product>(
    $"product:{id}",
    _ => GetProductFromDb(id),
    options
);
```

🚀 Hybrid != L1+L2

Il nostro codice rimane sempre **lo stesso**.

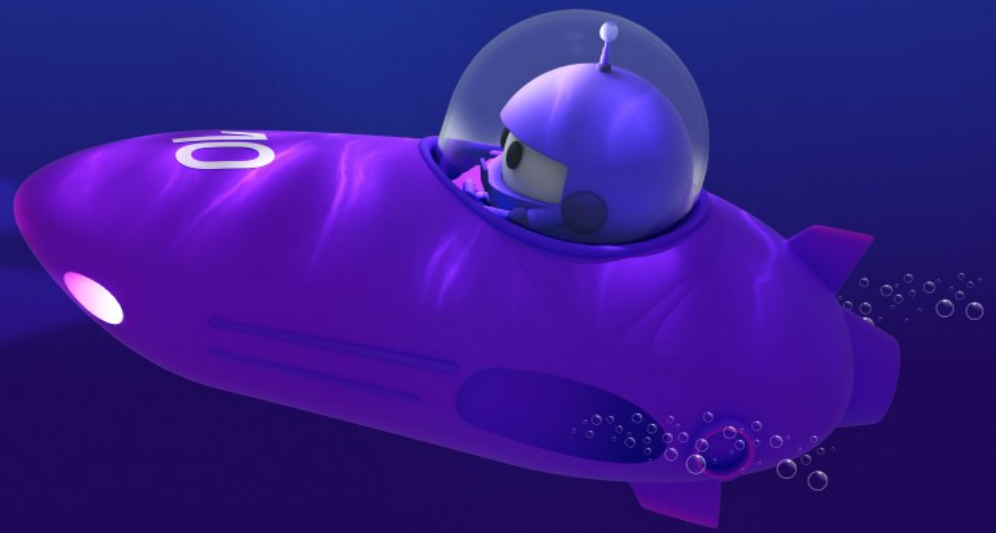
Non c'è bisogno di cambiarlo ovunque, solo una riga durante il **setup**.

Scenari:

- **L1** sviluppo locale ➡️ **L1+L2** in staging/produzione
- **L1** inizialmente (pochi utenti) ➡️ **L1+L2** quando arriva il successo (dobbiamo scalare)

Tutto **senza toccare** il nostro codice.

Una Hybrid Cache != HybridCache



🤔 Una Hybrid Cache != HybridCache

Oh, un'ultima cosa.

Quando diciamo **"una memory cache"** ci riferiamo al **tipo di cache**, in generale.

Non intendiamo necessariamente `MemoryCache` di Microsoft, giusto?

Ad esempio, `BitFaster.Caching` è anch'essa "una memory cache".

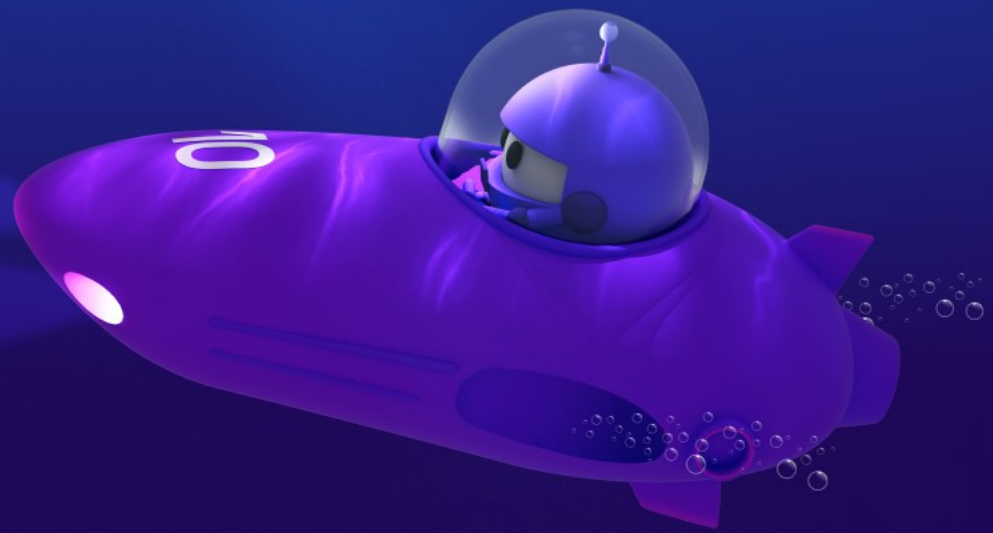
🤔 Una Hybrid Cache != HybridCache

Allo stesso modo, quando diciamo **"una hybrid cache"** ci riferiamo al **tipo di cache**, in generale.

Non intendiamo necessariamente `HybridCache` di Microsoft (2025).

Ad esempio, `FusionCache` (2020) è anch'essa "una hybrid cache".

Librerie



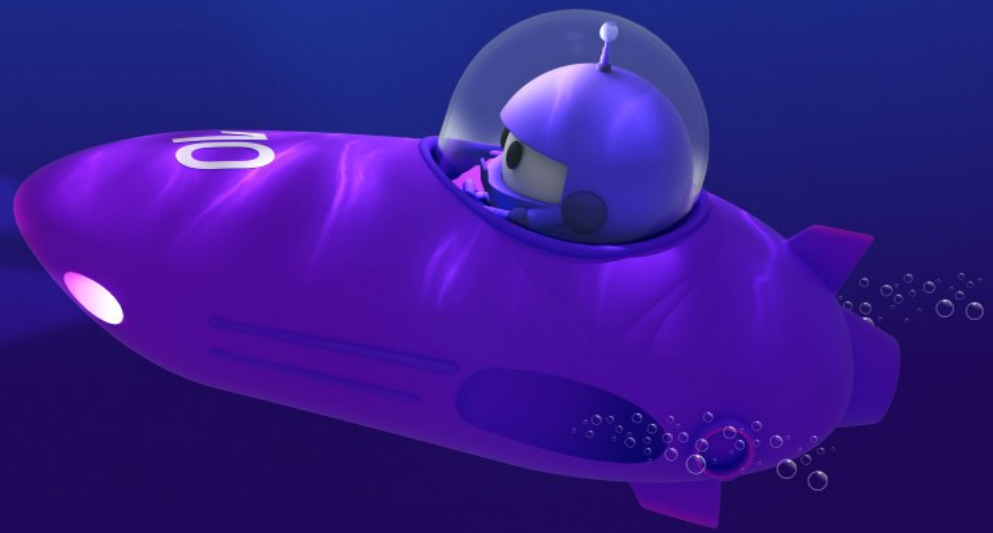
📦 Librerie

In questa sessione ci concentreremo su:

FusionCache

HybridCache

FusionCache



FusionCache

Free + OSS

Easy to use, fast and robust hybrid cache with advanced resiliency features.

github.com/ZiggyCreatures/FusionCache

⬇️ 36M download (15M)
⭐ 3.3K
📜 MIT License
💰 Free

FusionCache

Avendo lavorato con la maggior parte dei tipi di cache, ho affrontato molti dei **problemi reali** che emergono quando si lavora con caching.

E ho visto come prevenirli, quando possibile, e come mitigarli/risolverli, quando inevitabili.

La **comunità OSS** mi ha sempre dato molto: a volte ho contribuito un po' (patch, bug fix, etc) ma mai in modo significativo.

Così ho deciso di provare a fare di più, e nel 2020 è nato FusionCache.

FusionCache

Il design:

- **L1:** usa `IMemoryCache`
- **L2:** usa `IDistributedCache`

Qualsiasi implementazione di `IDistributedCache` funziona: Redis, Memcached, SQLite, MongoDB, etc (lista completa in online doc).

Gestisce in modo trasparente:

- uno o due **livelli**
- una o più **istanze/nodi**

senza cambiare nemmeno una riga di codice.

FusionCache

Setup (dopo aver installato il **pacchetto**):

```
services.AddFusionCache();
```

ℹ️ **NOTA:** si può anche fare direttamente `new FusionCache()` (non scontato...)

FusionCache

Si può anche **configurare** molto di più, grazie ad un **fluent builder**:

```
services.AddFusionCache()
    .WithOptions(...)
    .WithDistributedCache(...)
    .WithSerializer(...)
    .WithBackplane(...);
```

FusionCache

Come **usarla**:

```
public class ProductController : Controller
{
    IFusionCache _cache;

    public ProductController(IFusionCache cache)
    {
        _cache = cache;
    }

    [HttpGet("{id}")]
    public ActionResult<Product> Get(int id)
    {
        return _cache.GetOrSet(
            $"product:{id}",
            _ => GetProduct(id),
            options => options.SetDuration(TimeSpan.FromMinutes(5))
        );
    }
}
```

FusionCache

Come **usarla** :

```
public class ProductController : Controller
{
    IFusionCache _cache;

    public ProductController(IFusionCache cache)
    {
        _cache = cache;
    }

    [HttpGet("{id}")]
    public ActionResult<Product> Get(int id)
    {
        return _cache.GetOrSet(
            $"product:{id}",
            _ => GetProduct(id),
            options => options.SetDuration(TimeSpan.FromMinutes(5))
        );
    }
}
```

FusionCache

Come **usarla** :

```
public class ProductController : Controller
{
    IFusionCache _cache;

    public ProductController(IFusionCache cache)
    {
        _cache = cache;
    }

    [HttpGet("{id}")]
    public ActionResult<Product> Get(int id)
    {
        return _cache.GetOrSet(
            $"product:{id}",
            _ => GetProduct(id),
            options => options.SetDuration(TimeSpan.FromMinutes(5))
        );
    }
}
```

FusionCache

Di base:

- il target è .NET Standard 2.0 (**ovunque**: vecchio .NET + nuovo .NET)
- è completamente **sync** + **async** (nessun sync-over-async)
- ricco set di opzioni: globali + entry + `DefaultEntryOptions` + ereditarietà

FusionCache ha molte **feature** per gestire **esigenze** e **problemi** reali.

 FusionCache

Vedremo di più nella **seconda sessione**, ma per avere un'idea:

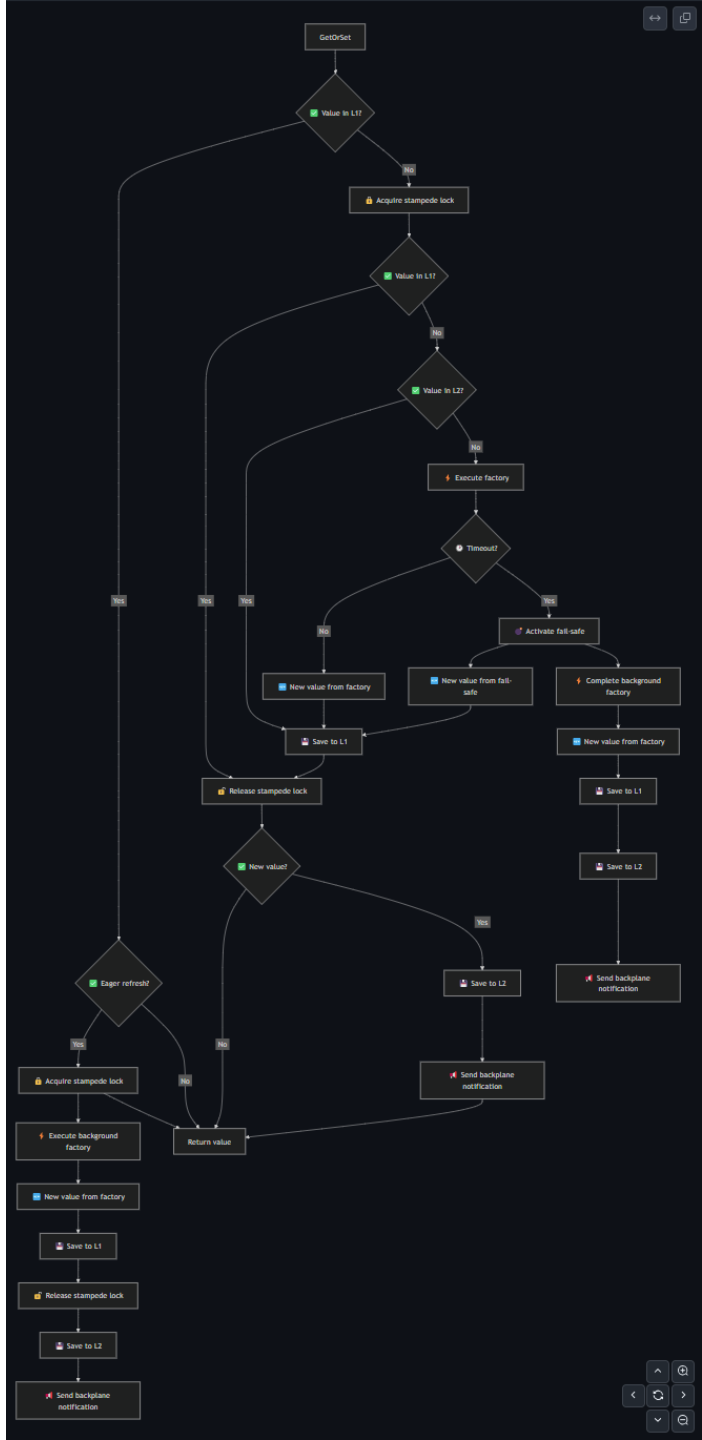
- **Cache Stampede:** protezione
- **Fail-Safe:** problemi temporanei con il database
- **Eager Refresh:** rallentamenti temporanei con il database
- **Factory Timeout:** rallentamenti temporanei con il database
- **Backplane:** notifiche istantanee su più istanze/nodi
- **Named Caches:** cache multiple, come HTTP Named Client ma per le cache
- **Tagging:** gestione gruppi/referenze
- **Auto-Recovery:** self-healing delle parti distribuite
- **Observability:** log, traces, metrics (supporto OTEL nativo)

FusionCache

Un sacco di documentazione, sia **inline** (IntelliSense) che **online**:

- introduzioni
- ogni feature, con design e dietro le quinte delle decisioni
- esempi
- diagrammi di flusso
- step by step (serve un po' di caffè)

Sì, tengo molto alla documentazione



FusionCache

Ok, ma qualcuno usa FusionCache?

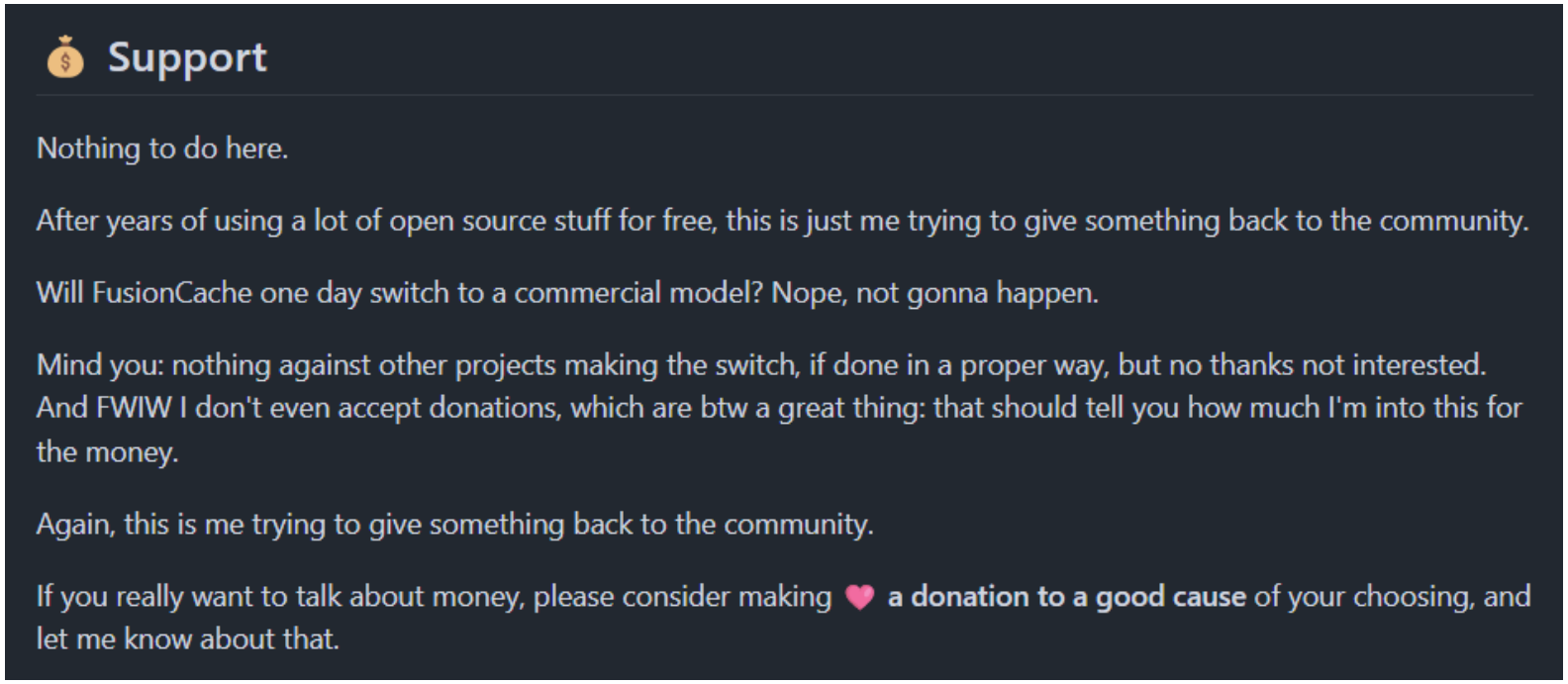
Un sacco di progetti, sia privati che OSS.

Molte aziende, dalle piccole alle decisamente grosse come **Have I Been Pwned**, **Dometrain** e persino **Microsoft** stessa.

E a proposito di Microsoft: FusionCache è il motore di cache in **Data API Builder** (DAB).

FusionCache

FusionCache diventa a pagamento? No. Nada. Nein. Nope.



Support

Nothing to do here.

After years of using a lot of open source stuff for free, this is just me trying to give something back to the community.

Will FusionCache one day switch to a commercial model? Nope, not gonna happen.

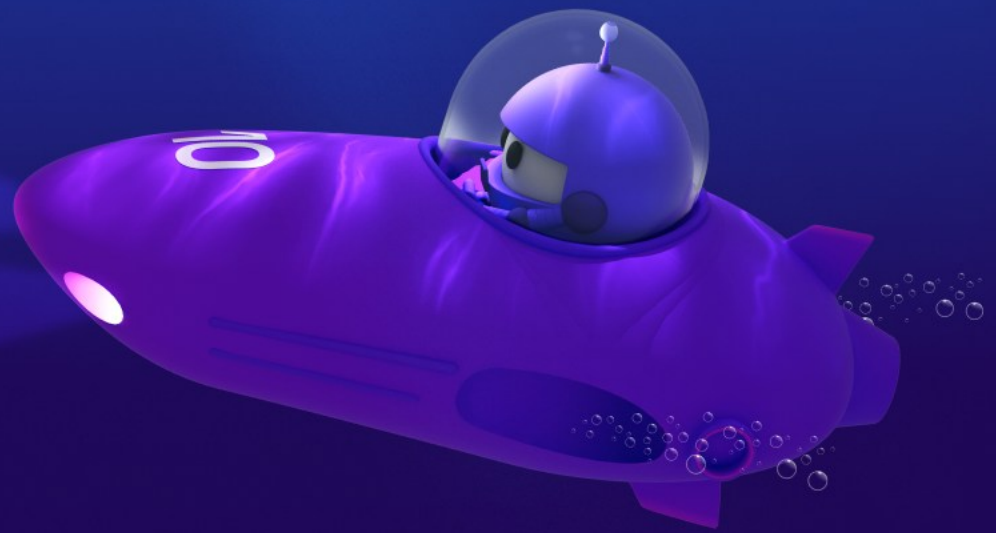
Mind you: nothing against other projects making the switch, if done in a proper way, but no thanks not interested. And FWIW I don't even accept donations, which are btw a great thing: that should tell you how much I'm into this for the money.

Again, this is me trying to give something back to the community.

If you really want to talk about money, please consider making 💖 **a donation to a good cause** of your choosing, and let me know about that.

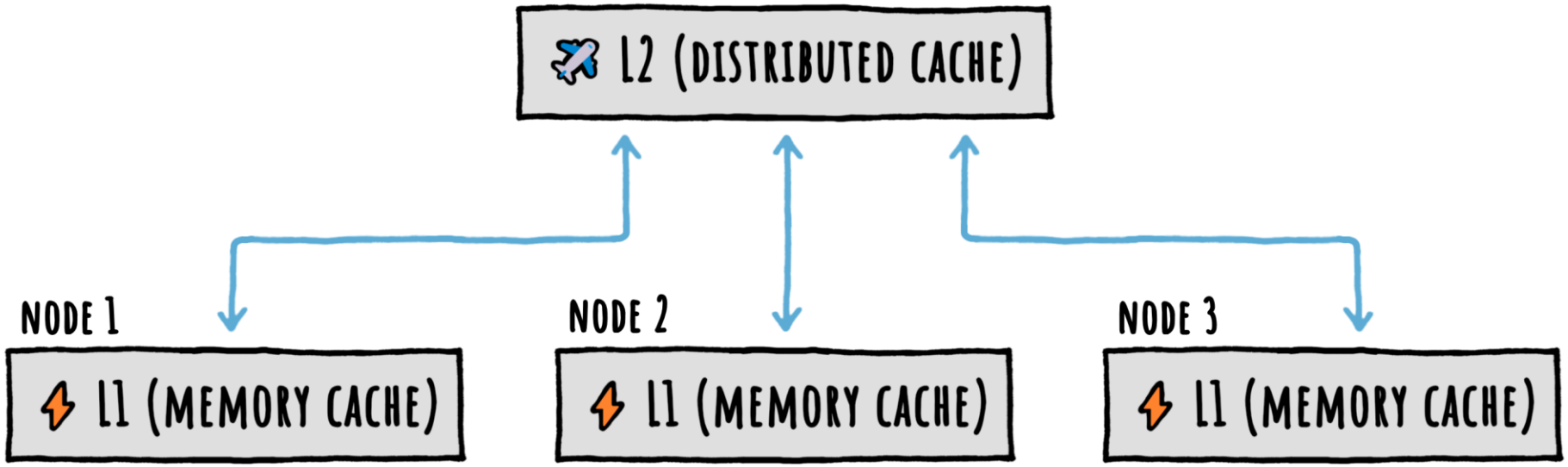
Sto solo **restituendo** un po' di ciò che ho **già ricevuto** dalla community OSS.

Cache Coherence



😨 Cache Coherence

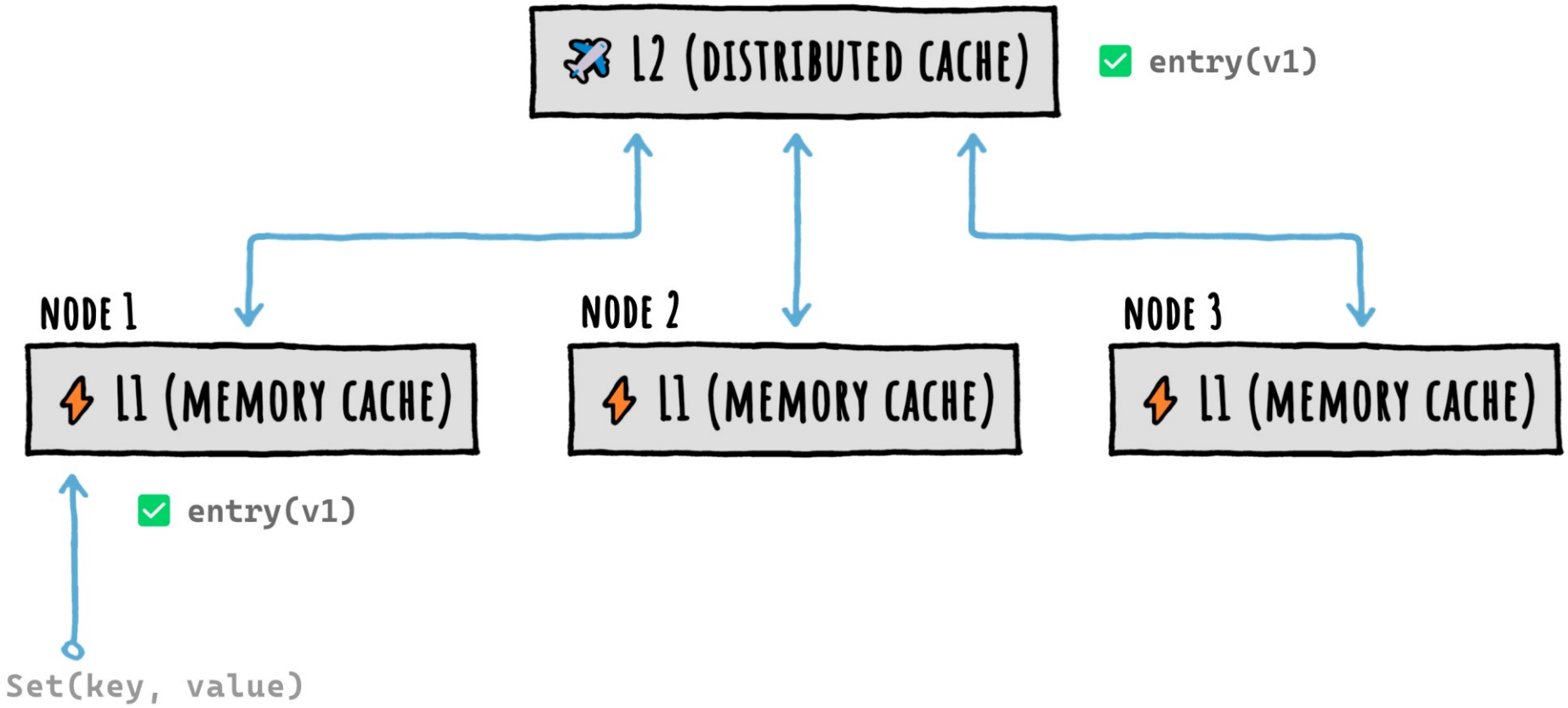
Supponiamo di essere in un setup **L1+L2** su un ambiente **multi-nodo**:



😨 Cache Coherence

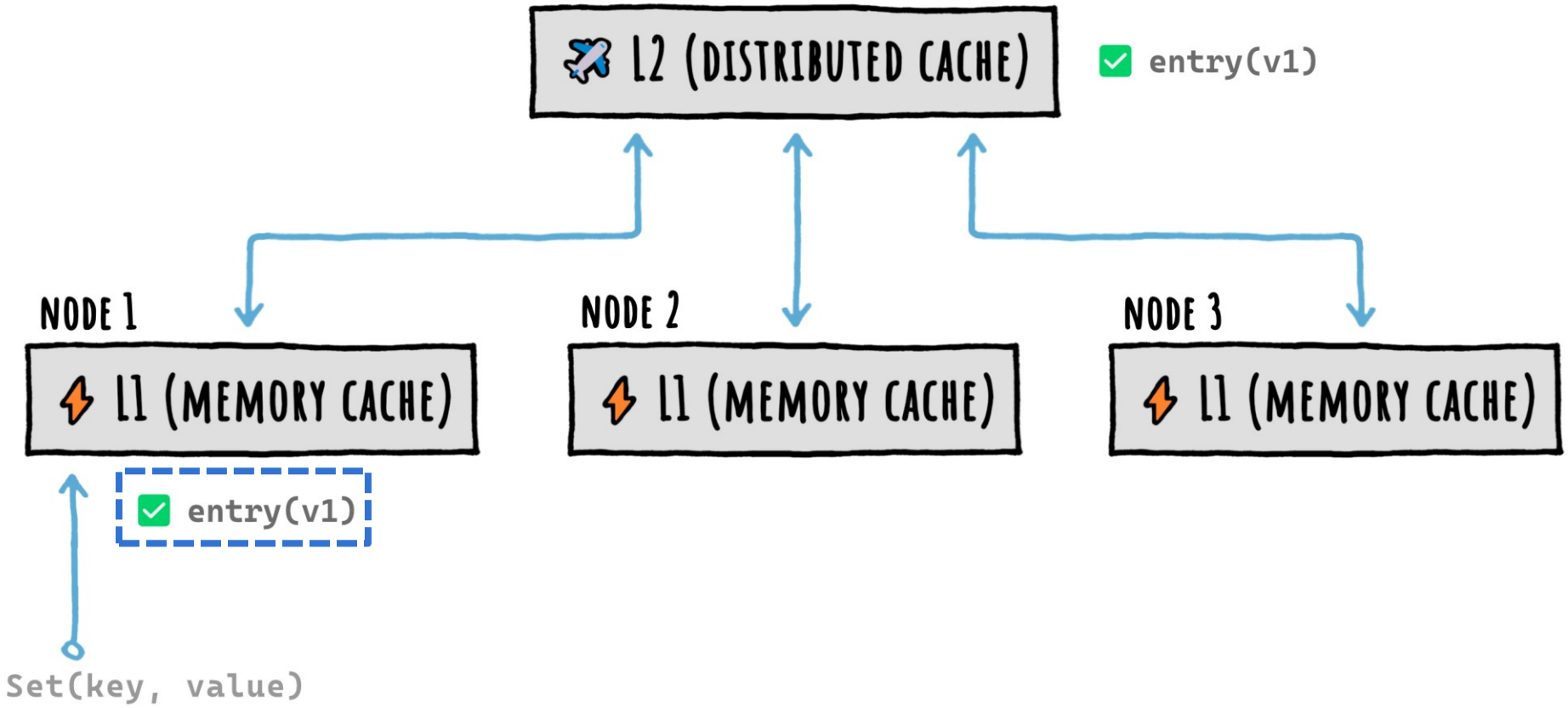
Cosa succede quando un nodo **modifica** una entry?

La hybrid cache scrive sia su **L1+L2**:



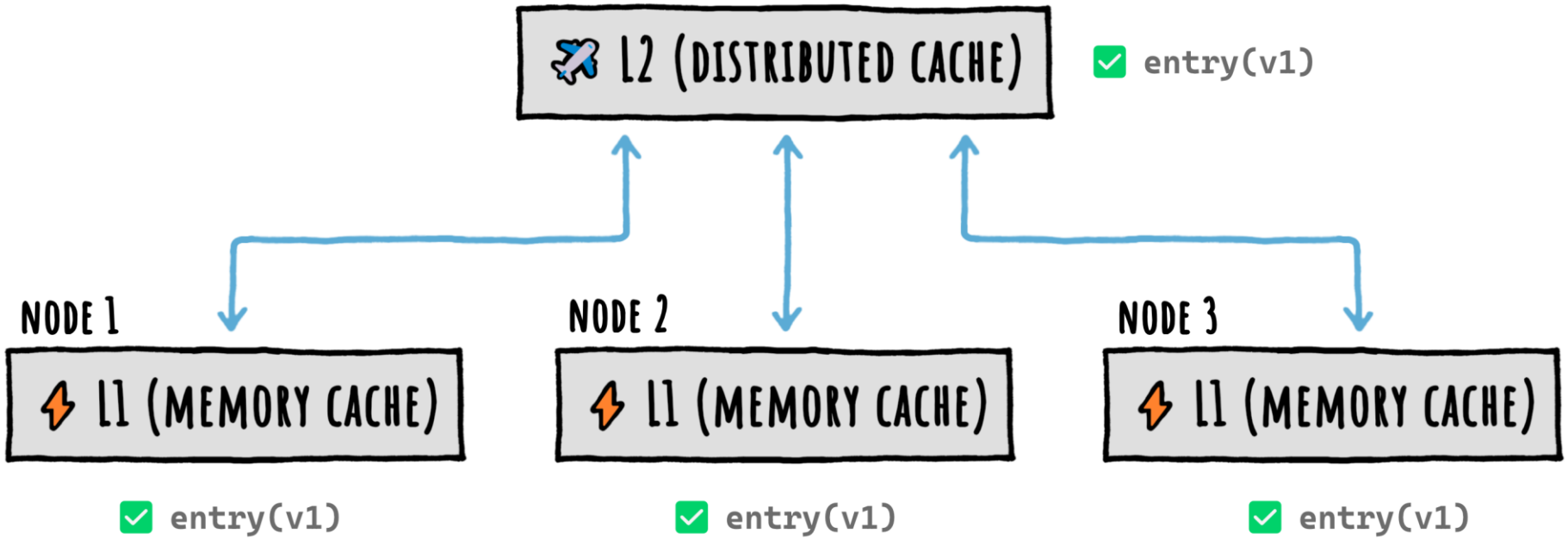
😨 Cache Coherence

... ma riguardo ad **L1**, viene aggiornata **solo** dove è stata eseguita l'operazione:



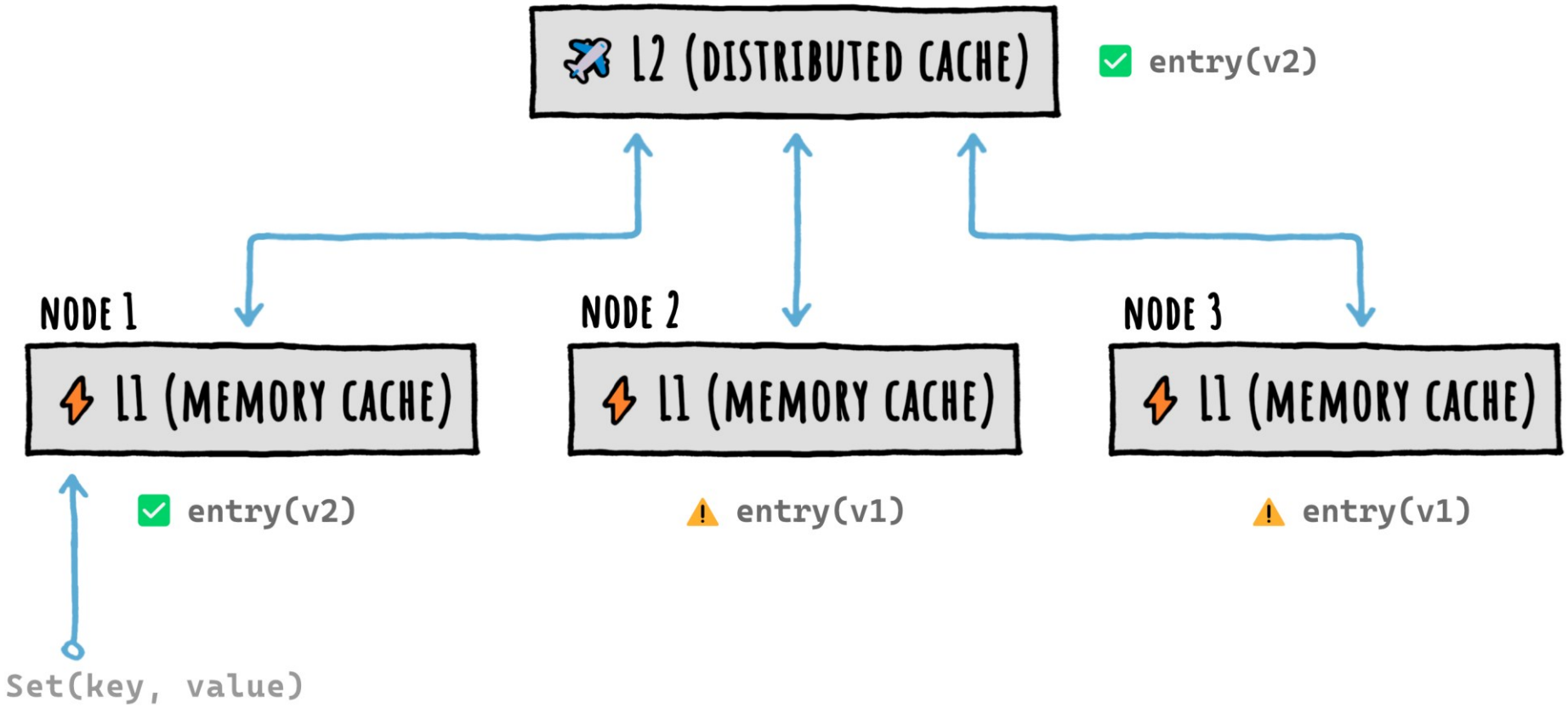
😨 Cache Coherence

E se gli altri nodi **avessero già** quel dato nella **loro L1** (memory)?



😨 Cache Coherence

Succederebbe che le **L1** negli **altri nodi** resterebbero col **vecchio valore** in cache:



😨 Cache Coherence

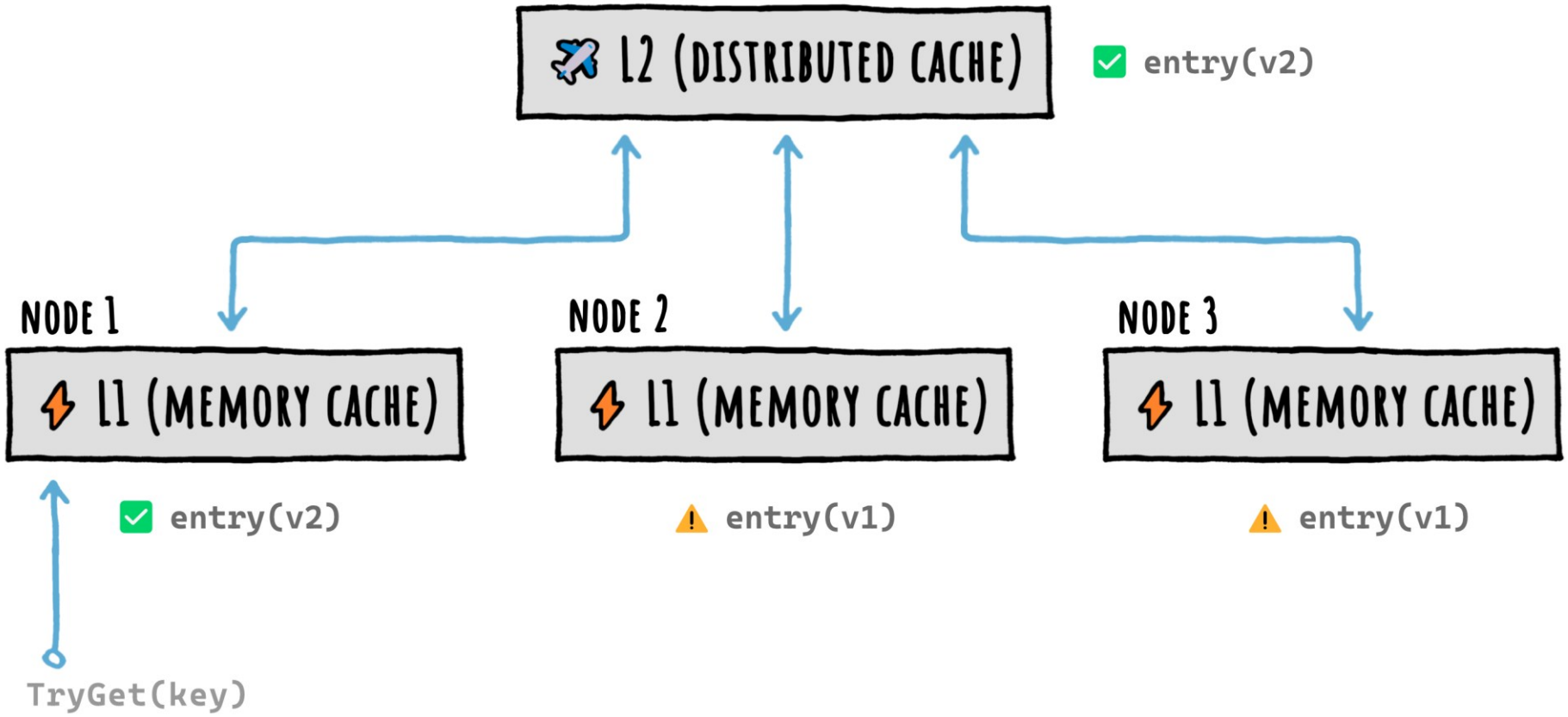
A quel punto dovremmo aspettare che il dato scada normalmente tramite **expiration**.

E nel mentre?

Cosa avviene **dopo** l'update ma **prima** della expiration?

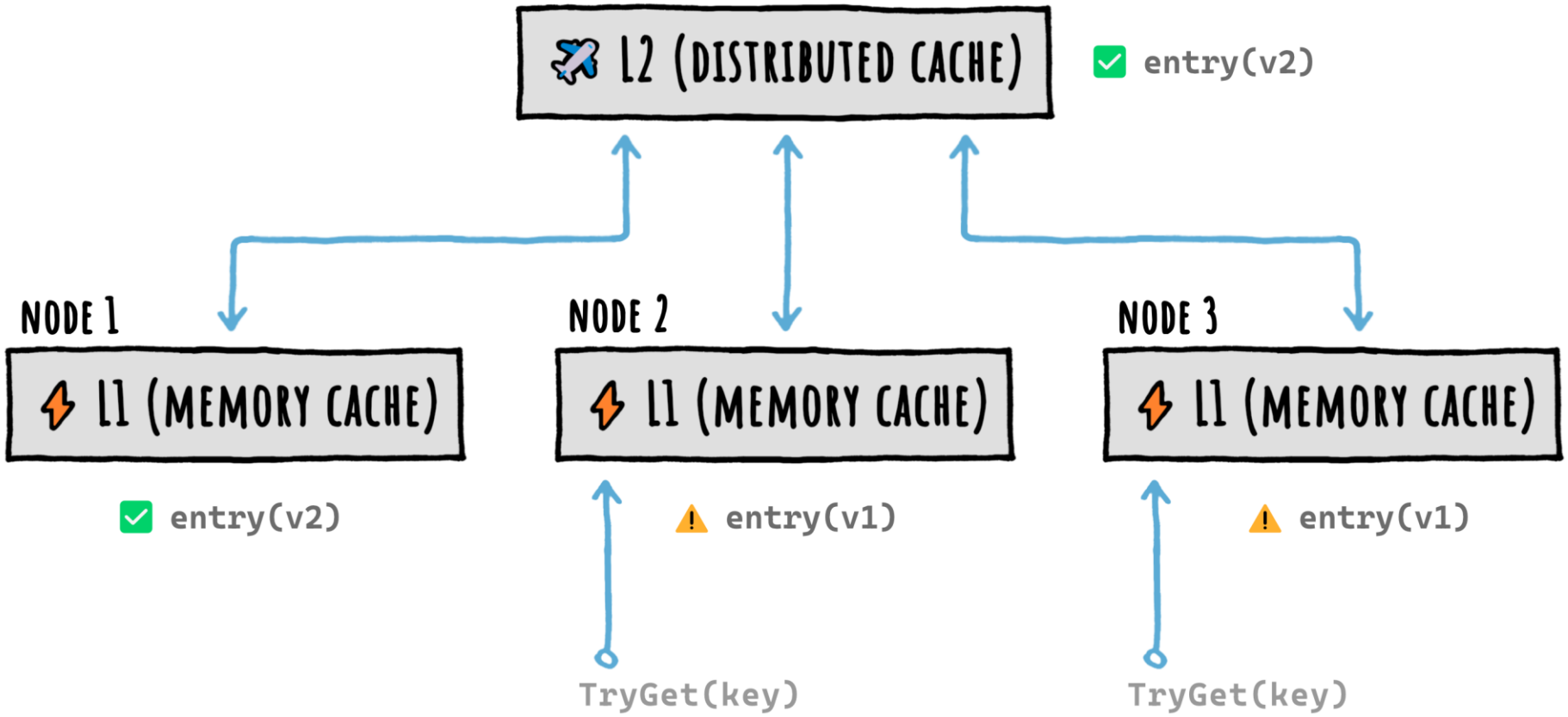
😨 Cache Coherence

Una richiesta servita dal **nodo 1** ritornerebbe il **valore nuovo**:



😨 Cache Coherence

Mentre una richiesta servita dal **nodo 2 o 3** ritornerebbe il **valore vecchio**:



😨 Cache Coherence

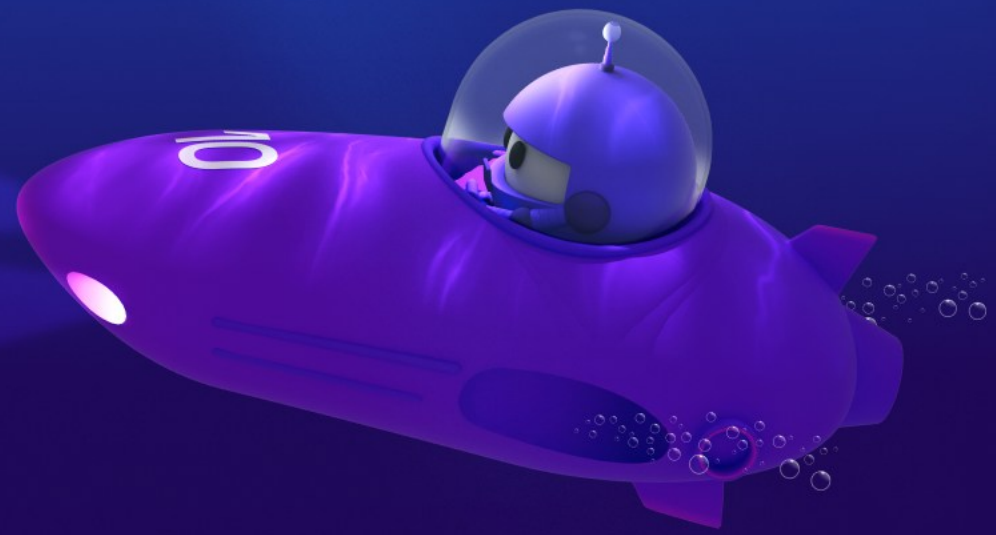
Fondamentalmente, la cache nel **suo insieme** diventa **incoerente**.

Ossia fornisce, per la **stessa domanda** e nello **stesso momento**, risposte **diverse**

E questo è **grave**.

Cosa possiamo fare?

Backplane (FusionCache)



📢 Backplane

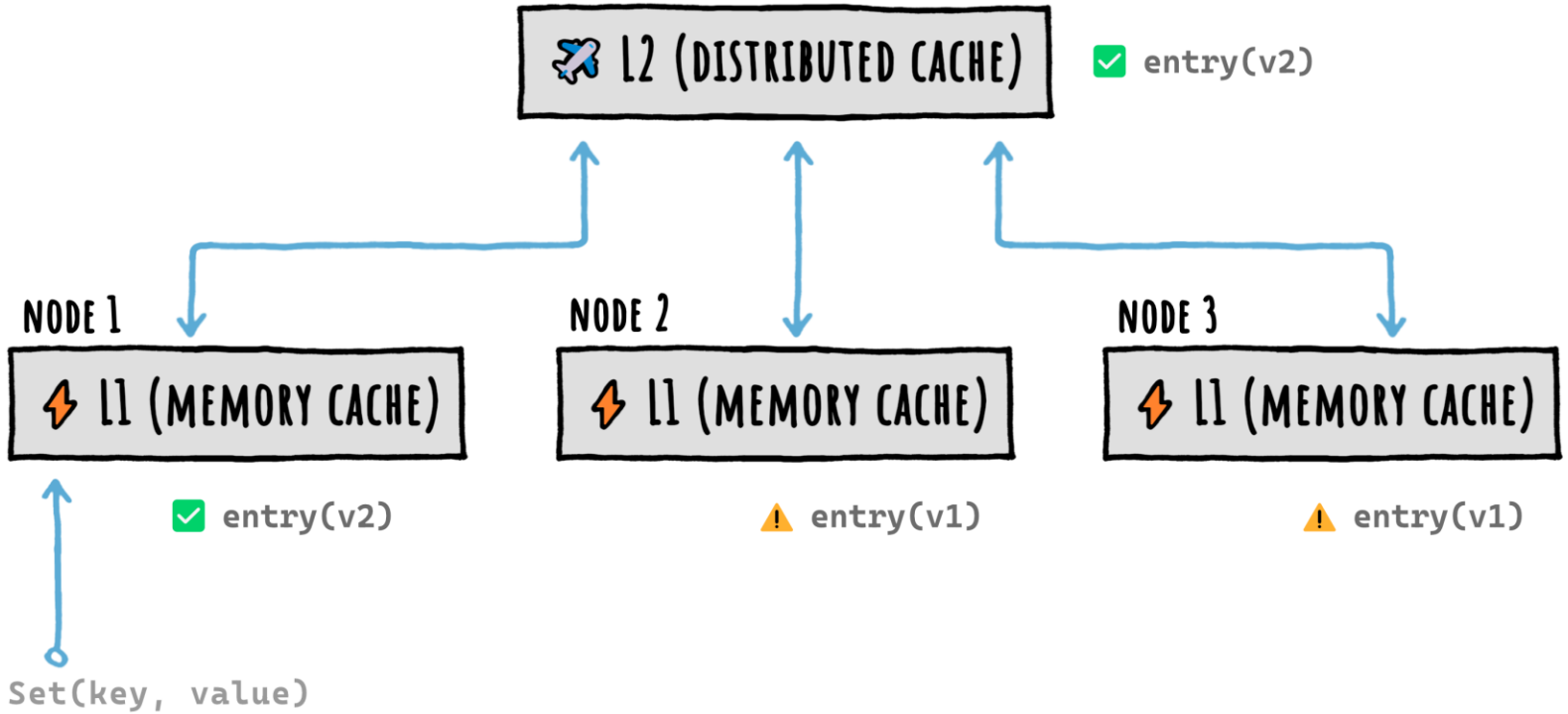
Possiamo semplicemente usare un **Backplane** per far comunicare i nodi fra di loro:

```
services.AddFusionCache()
  .WithSerializer(
    new FusionCacheSystemTextJsonSerializer()
  )
  .WithDistributedCache(
    new RedisCache(new RedisCacheOptions { ... })
  )
  // BACKPLANE
  .WithBackplane(
    new RedisBackplane(new RedisBackplaneOpsions { ... })
  );
```

Fatto.

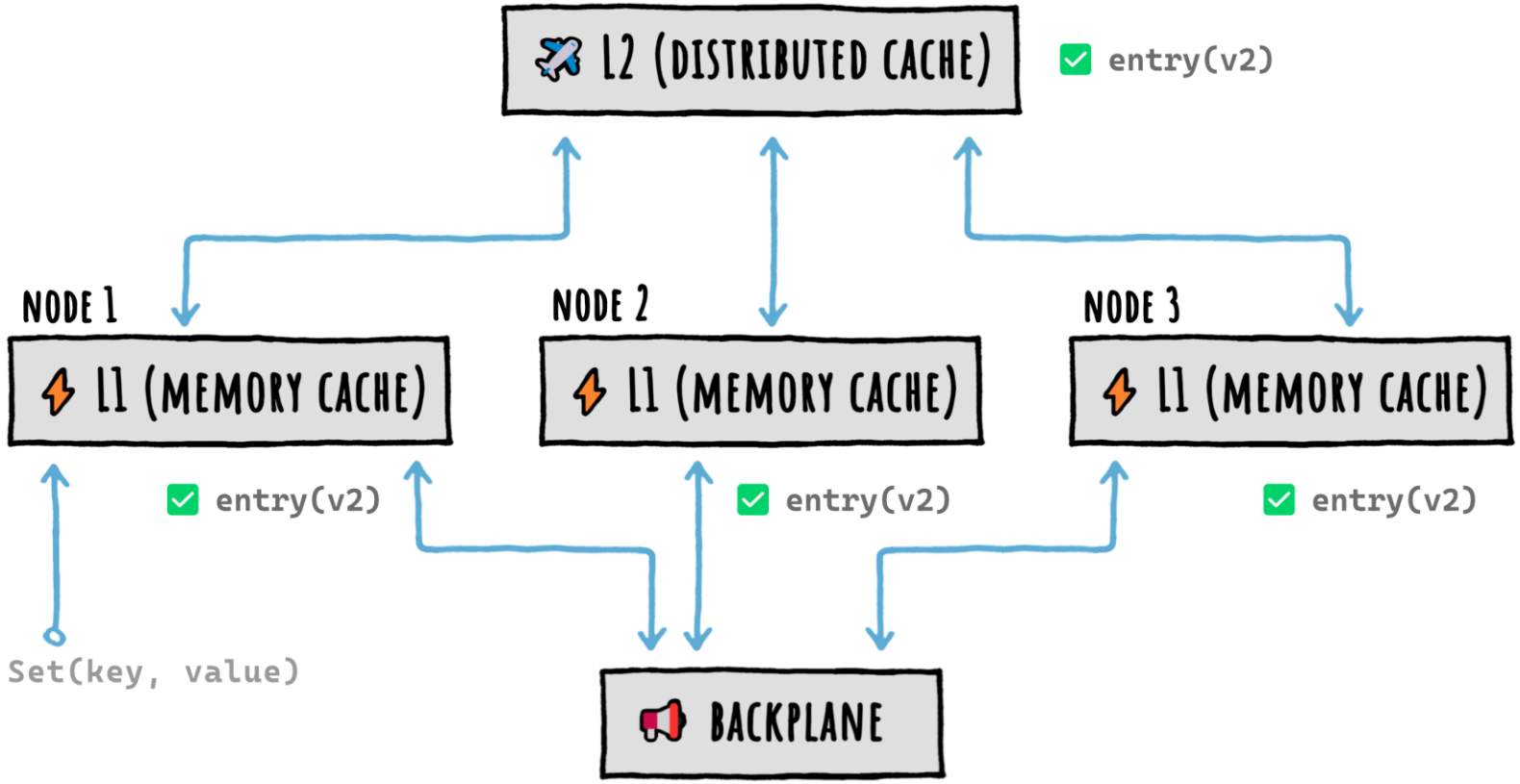
📢 Backplane

Passando così da una cache **incoerente**:

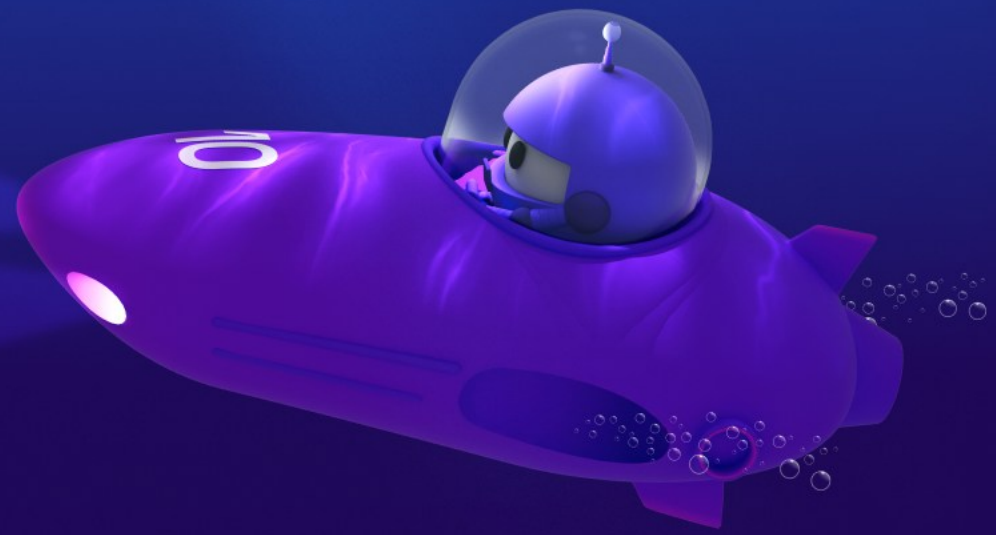


📢 Backplane

Ad una cache **sempre coerente**, senza dover modificare il nostro codice:



Microsoft HybridCache



HybridCache

All'inizio del 2024, **Microsoft** ha annunciato la propria **HybridCache**.

Probabilmente ne avete già sentito parlare, vero?

Beh, come direbbe **Íñigo Montoya**: non credo che significhi quello che pensi tu.

Vediamo...

HybridCache

Il design:

- **L1:** usa `IMemoryCache`
- **L2:** usa `IDistributedCache`

Qualsiasi implementazione di `IDistributedCache` funziona: Redis, Memcached, SQLite, MongoDB, etc (lista completa in online doc).

Gestisce in modo trasparente:

- uno o due **livelli**
- una o più **istanze/nodi (*)**

Design di FusionCache: validato 🥳

HybridCache

Di base:

- il target è .NET Standard 2.0 (**ovunque**: vecchio .NET + nuovo .NET)
- set di opzioni: globali + entry + `DefaultEntryOptions`

Feature:

- **Cache Stampede (*):** protezione
- **Tagging (*):** gestione gruppi/referenze

HybridCache

Ma soprattutto, è sia una 1st party **shared abstraction** che una **default implementation**.

Infatti:

- **abstraction:** `public abstract class HybridCache` (.NET 9)
- **implementation:** `internal class DefaultHybridCache` (.NET Extensions)

Avere una astrazione significa poter avere altre implementazioni.

HybridCache

Possiamo pensare alla classe astratta `HybridCache` come all'interfaccia `IDistributedCache`.

Ossia:

- l'interfaccia `IDistributedCache` è una astrazione per generiche cache **distribuite**
- la classe astratta `HybridCache` è una astrazione per generiche cache **ibride**

E questo apre, come vedremo, scenari interessanti.

HybridCache

Setup (dopo aver installato il **pacchetto**):

```
services.AddHybridCache();
```

HybridCache

Come **usarla** :

```
public class ProductController : Controller
{
    HybridCache _cache;

    public ProductController(HybridCache cache)
    {
        _cache = cache;
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<Product>> Get(int id)
    {
        return await _cache.GetOrCreateAsync<Product>(
            $"product:{id}",
            async _ => GetProduct(id),
            options
        );
    }
}
```

HybridCache

Come **usarla** :

```
public class ProductController : Controller
{
    HybridCache _cache;

    public ProductController(HybridCache cache)
    {
        _cache = cache;
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<Product>> Get(int id)
    {
        return await _cache.GetOrCreateAsync<Product>(
            $"product:{id}",
            async _ => GetProduct(id),
            options
        );
    }
}
```

HybridCache

Ho condiviso con il team suggerimenti, idee, criticità e... **hanno ascoltato**: fantastico 🥳

Grazie Marc Gravell e team!

Credo che lo sforzo fatto con HybridCache sia un ottimo esempio di come potrebbe essere quando **Microsoft** e la **comunità OSS** hanno un dialogo costruttivo.

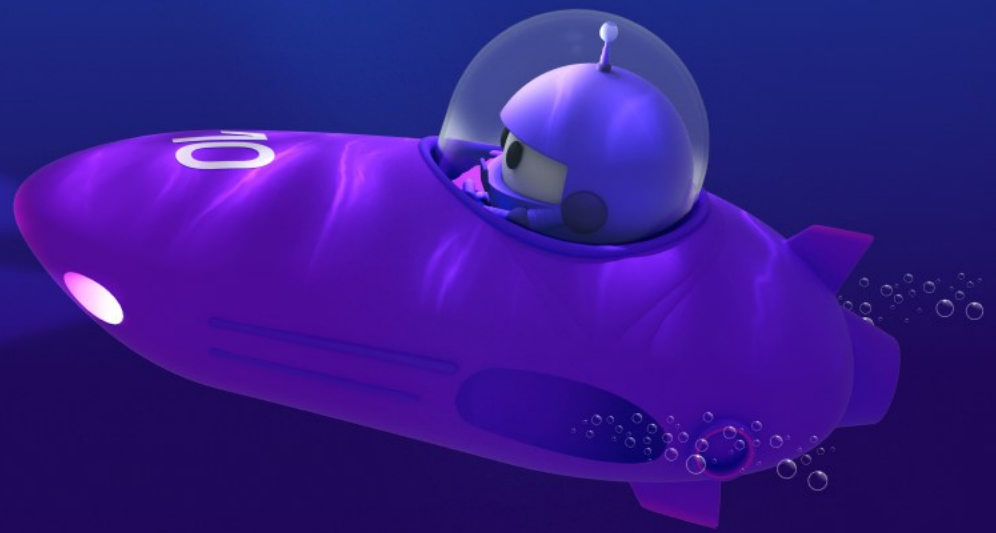
HybridCache

Quindi, state già usando HybridCache?

Magari in **produzione**?

Ok, fate attenzione.

HybridCache: Limitazioni E Problemi (fine 2025)



HybridCache: Limitazioni E Problemi

Poiché l'attuale implementazione Microsoft è la primissima **versione**, presenta ancora alcune limitazioni e problemi.

La maggior parte **non** è legata all'**astrazione**, ma **solo** alla attuale **implementazione di default**.

Alcuni sono minori, altri più gravi, quindi è importante conoscerli.

Vediamo.

HybridCache: Limitazioni E Problemi

Ad oggi:

- impossibilità di istanziazione diretta, solo DI
- con DI, no controllo su L1/L2
- con DI, singola istanza
- async only
- no metodi read-only
- non deterministica su cache miss
- incoerente con più nodi

Capiamo meglio questi punti?

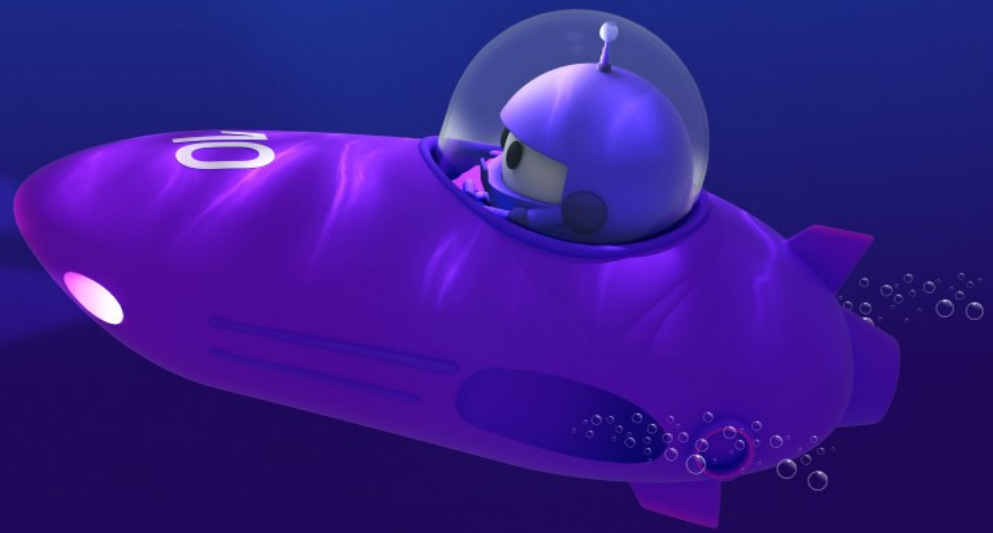
Vediamo come risolverli?

HybridCache: Limitazioni E Problemi

Si ma dopo pranzo, nella prossima sessione.

Ho fame 🙃

Recap



⚡ Le Memory Cache

Pro/Contro:

- ✅ **facile:** facile da usare, meno codice
- ✅ **data locality:** vicino al nostro stesso codice, nello stesso spazio di memoria
- ✅ **costo:** nessuna network call, nessuna (de)serializzazione
- ✅ **availability:** è sempre disponibile
- ✅ **stampede protection:** spesso (dipende dalla caching library)
- ⛔ **cold start:** ad ogni riavvio la cache è vuota
- ⛔ **scaling orizzontale:** i dati non sono condivisi tra più nodi

✈️ Le Distributed Cache

Pro/Contro:

- ⛔ **facile:** meno facile da usare, gestione manuale (de)serializzazione, etc
- ⛔ **data locality:** i dati sono fuori processo, tipicamente remoti
- ⛔ **costo:** network call + (de)serializzazione
- ⛔ **availability:** potrebbe non essere sempre disponibile (o raggiungibile)
- ⛔ **stampede protection:** nessuna
- ✅ **cold start:** ad ogni riavvio la cache è già popolata
- ✅ **scaling orizzontale:** i dati sono condivisi tra più nodi

🚀 Le Hybrid Cache

Pro/Contro:

✅ **facile:** facile da usare

✅ **data locality:** vicino al nostro stesso codice, nello stesso spazio di memoria (**L1**)

✅ **costo:** nessuna network call e (de)serializzazione (**L1**) tranne la prima L1 CACHE MISS

✅ **availability:** è sempre disponibile (**L1**)

✅ **stampede protection:** probabilmente (dipende dalla caching library)

✅ **cold start:** ad ogni riavvio L1 è vuoto, ma **L2** è già popolata

✅ **scaling orizzontale:** i dati sono condivisi tra più nodi (**L2**)

Inoltre: una **API unificata** e con più **feature**.

Il meglio di entrambi i mondi.

Grazie!

github.com/jodydonetti

twitter.com/jodydonetti

linkedin.com/in/jody-donetti

Su Dometrain:

Feedback, please 🥲