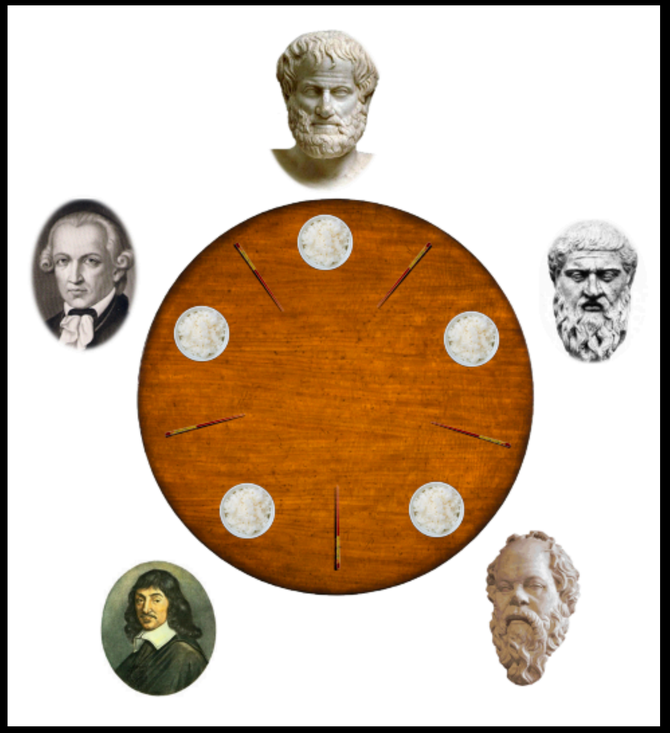
Plato ate my pasta

A cautionary tale about thread safety



Content

- Some code I wrote for network logger
- Why it blew up
- How I am fixing it

V.Quick description of dining philosophers

- TODO

Network Logger

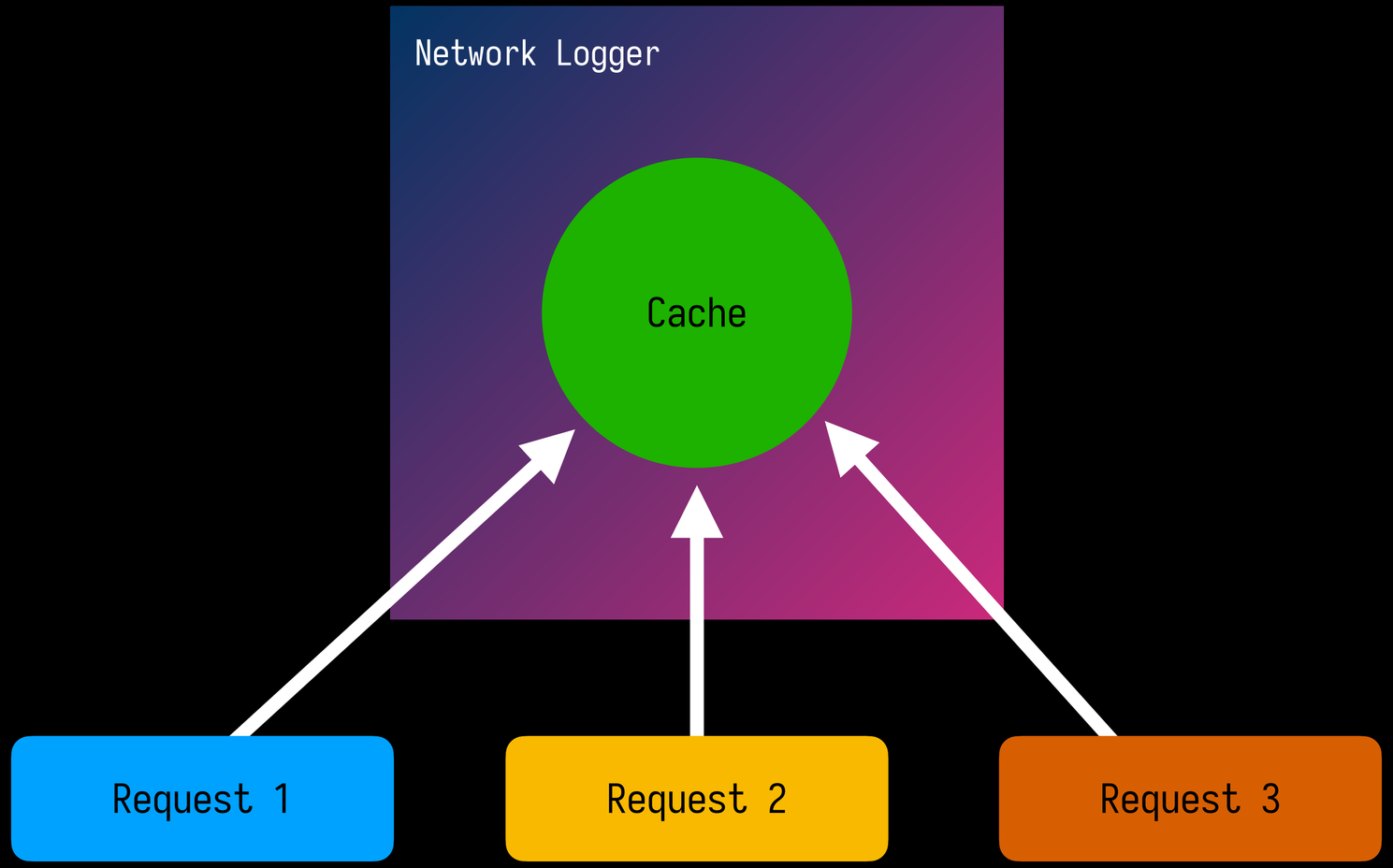
- So we have a cache of unique ids => a network log (req, response, time, code, etc)
    - `cache = Dictionary<UUID, NetworkLog>()`
- When the app sends a network request
    - `cache.add(networkRequest)`
- When 10 entries are added to the cache (the 10 is arbitrary-ish, json responses from our API can be large and we don’t want to hold them in memory forever)
    - `cache.writeToDiskAndDeleteAllFromCache()`

Hang on that doesn’t look *too* silly?

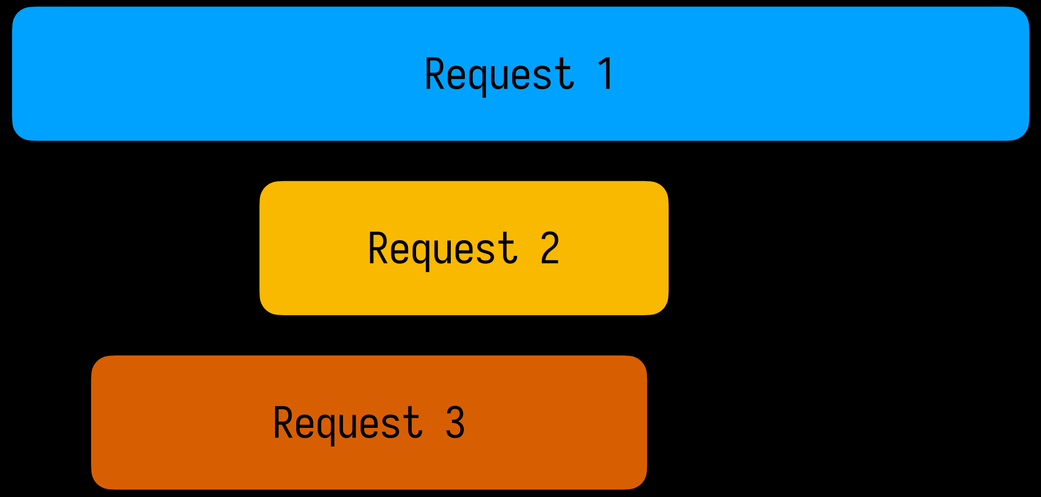
That’s what I thought

- However there was no guarantee about *where* this code was called from.
- It is invoked as part of the URL Protocol pipeline, which is highly async in nature.
- Reading + Writing across threads is a recipe for explosions.

Let’s follow the path to ruin



Not all tasks start and end at the same time



Dispatch Queues

FIFO Queue, for submitting tasks to a pool of threads

- Apple doesn’t trust us idiots to get concurrency through manual thread creation right
  - With good reason, barely anyone ever get’s it right
- So they wrote a set of training wheels for us called Grand Central Dispatch
- This provides an API for us to submit work items to a pool of threads.
- No guarantees about which thread your work will run on (bar the App Main thread)
- But it can guarantee work gets done according to how you submit the work.

You may have seen code like this before

```
someAsyncFunctionThatUsesABackgroundQueue { data in
  DispatchQueue.main.async { [weak self] in
      self?.myData = data
      self?.tableView.reloadData()
  }
}
```

Sync? Async?

- **Sync**: *Submits a work item for execution on the current queue and returns after that block finishes executing.*
- **Async**: *Schedules a work item for immediate execution, and returns immediately.*

Why async?

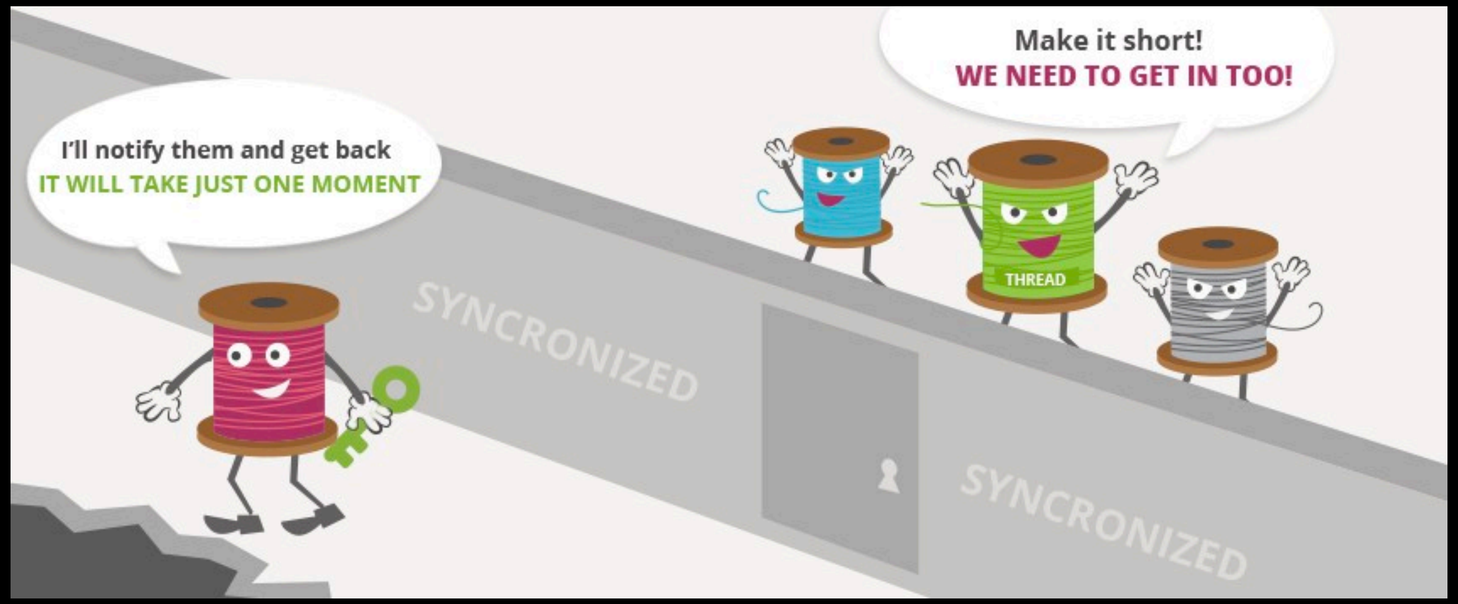
> **Important**
> Attempting to synchronously execute a work item on the main queue results in deadlock.

```
someAsyncFunctionThatUsesABackgroundQueue { data in
  DispatchQueue.main.async { [weak self] in
      myData = data
      self?.tableView.reloadData()
  }
}
```

Okay so how does this relate to our exploding dictionary?

I’m getting there, promise

- So now we know we have a process whereby we can control a threads access to resources (GCD).
- We now can give a task a consistent view of data the entire time it is interacting with a resource. (I think this isn’t actually true - this is the concept of atomicity - check with Tom)



Told ya I would get there

Impatient bunch…

```
public class ThreadSafeDictionary<K: Hashable, V>: Collection {
    private var dictionary: [K: V]
    private let queue = DispatchQueue(label: "com.trainline.threadsafedictionary", attributes: .concurrent)
    public subscript(key: K) -> V? {
        set(newValue) {
            queue.async(flags: .barrier) { [weak self] in
                self?.dictionary[key] = newValue
            }
        }
        get {
            queue.sync {
                return dictionary[key]
            }
        }
    }
```

Told ya I would get there

Impatient bunch…

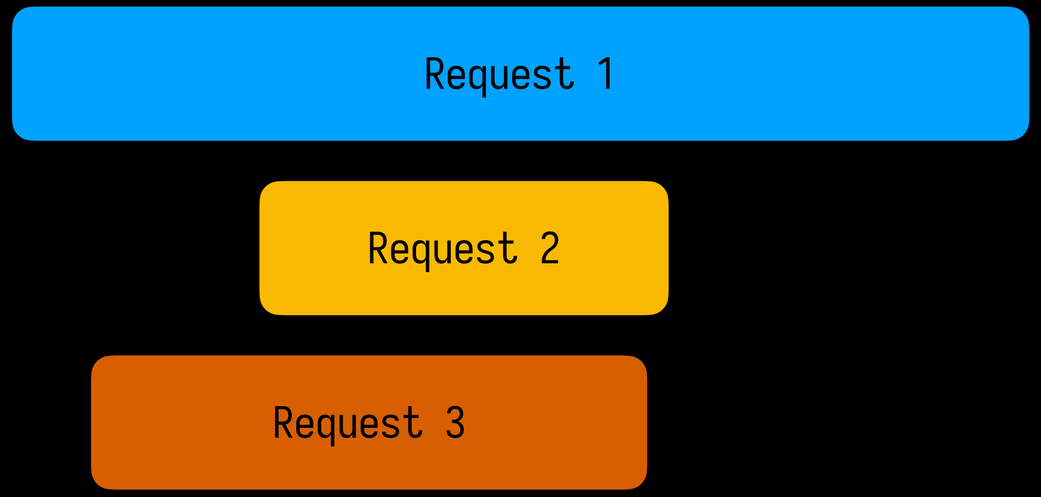
```
public class ThreadSafeDictionary<K: Hashable, V>: Collection {
    private var dictionary: [K: V]
    private let queue = DispatchQueue(label: "com.trainline.threadsafedictionary", attributes: .concurrent)
    public subscript(key: K) -> V? {
        set(newValue) {
            queue.async(flags: .barrier) { [weak self] in
                self?.dictionary[key] = newValue
            }
        }
        get {
            queue.sync {
                return dictionary[key]
            }
        }
    }
```

A non thread safe data structure

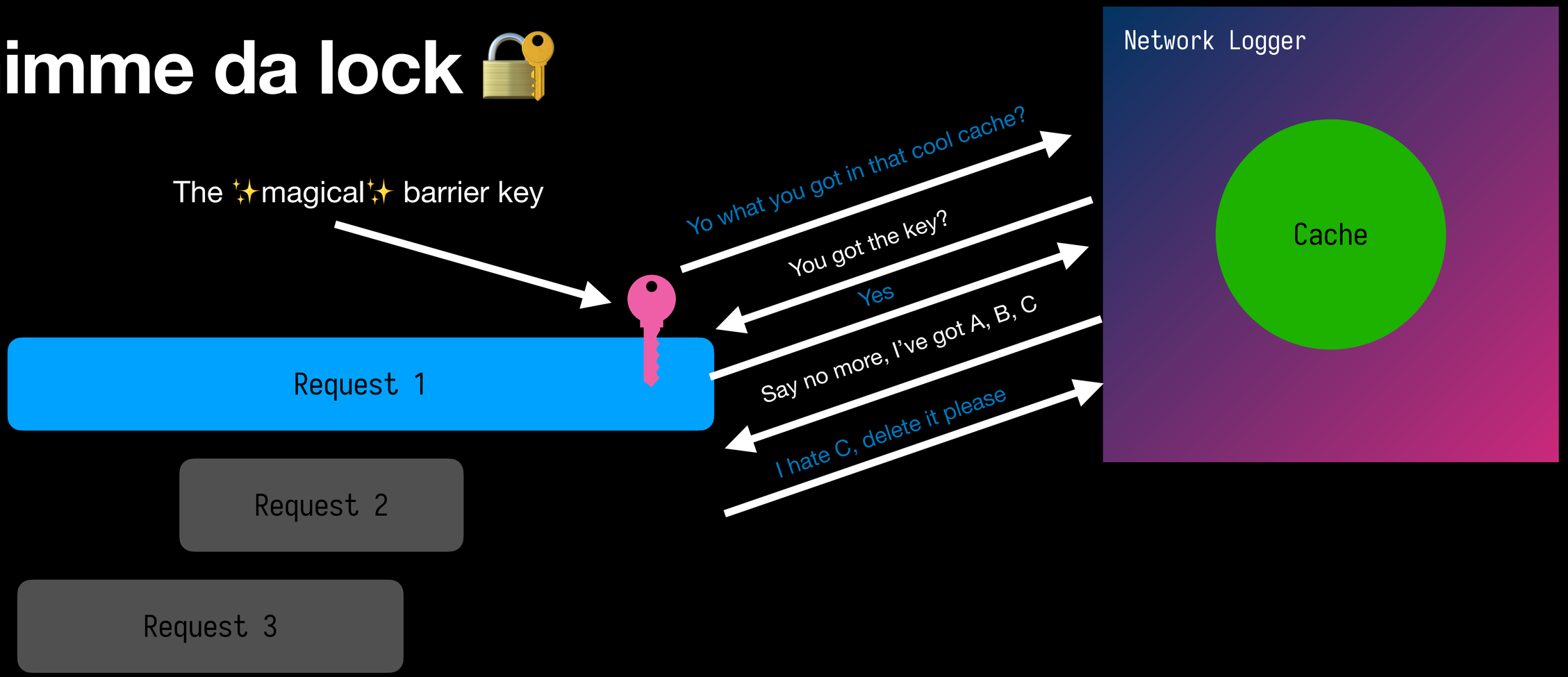
A concurrent serial queue

Controlled, access to the underlying data structure

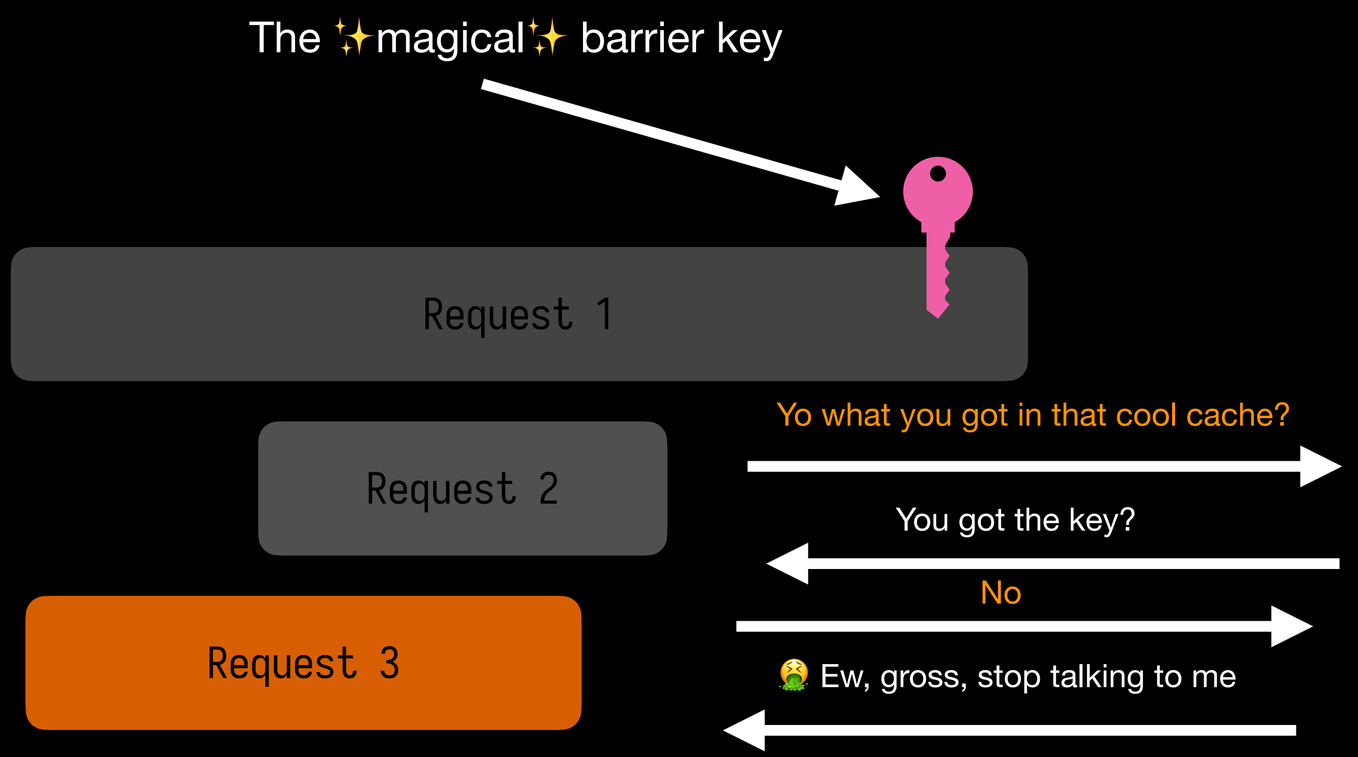
What does this change?

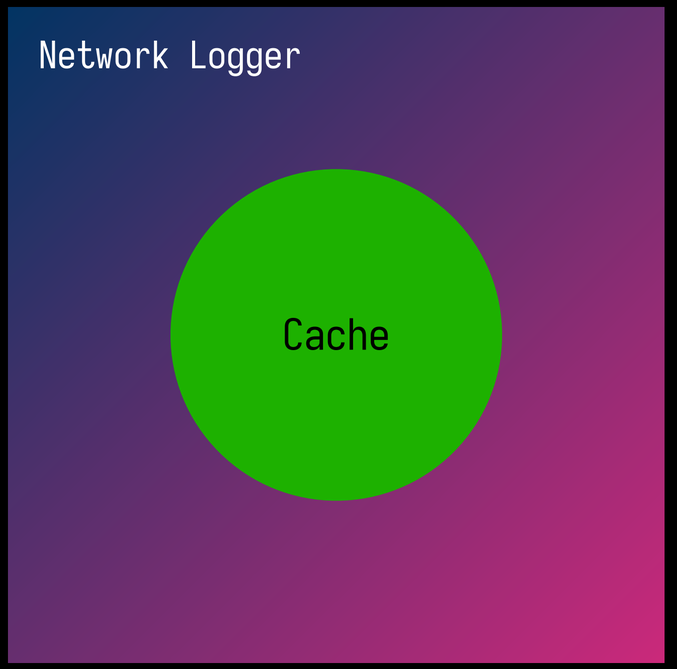


Gimme da lock 🔐



Our queue will now execute serially





Our queue will now execute serially