Performance Architecture of Dodo IS

Concurrency and Transient Fault Tolerance primitives

George Polevoy

Reliability Engineer, Dodo Pizza
g.polevoi@dodopizza.com

https://www.facebook.com/dodopizzaio/

ДОДО ПИЦЦА
Сеть пиццерий № 1 в России

Доставка пиццы Москва
Среднее время доставки 34 мин

8 800 302-00-60
Звонок бесплатный

Додо рубли ₽ | Войти

Пиццы | Комбо | Закуски | Десерты | Напитки | Акции | Контакты | Франшиза | О нас

0 ₽
1 товар

Корзина

Введите промокод | Применить

Dodo IS

Global

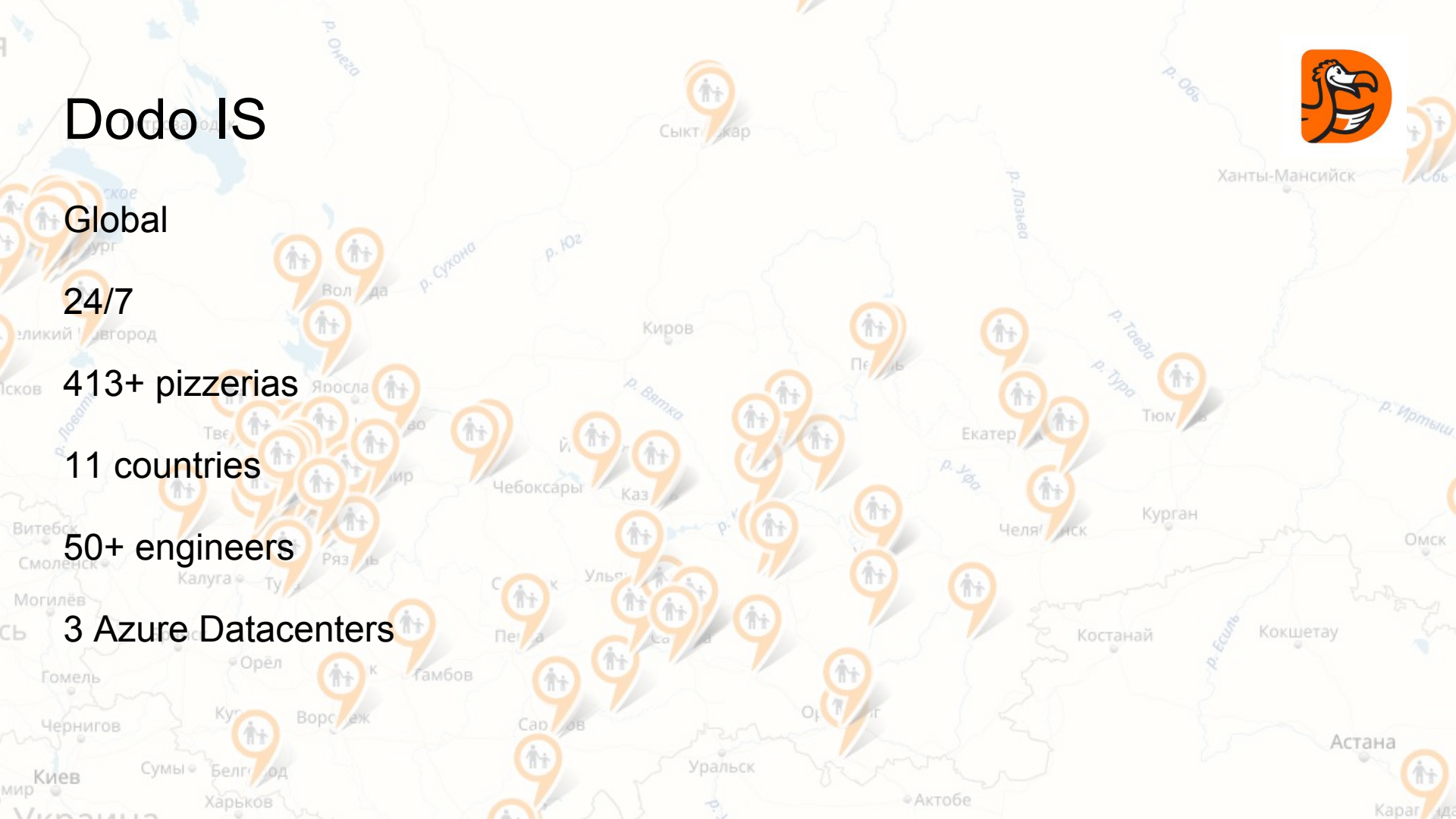
24/7

413+ pizzerias

11 countries

50+ engineers

3 Azure Datacenters



Dodo IS

Web, Mobile

Production

Delivery

Loyality

BI

HRM

ERP

Dodo IS

Scale

20+ services

100+ VMs

Dev infrastructure: ~20%

Monitoring infrastructure: ~10%

Technologies

ASP.NET + IIS

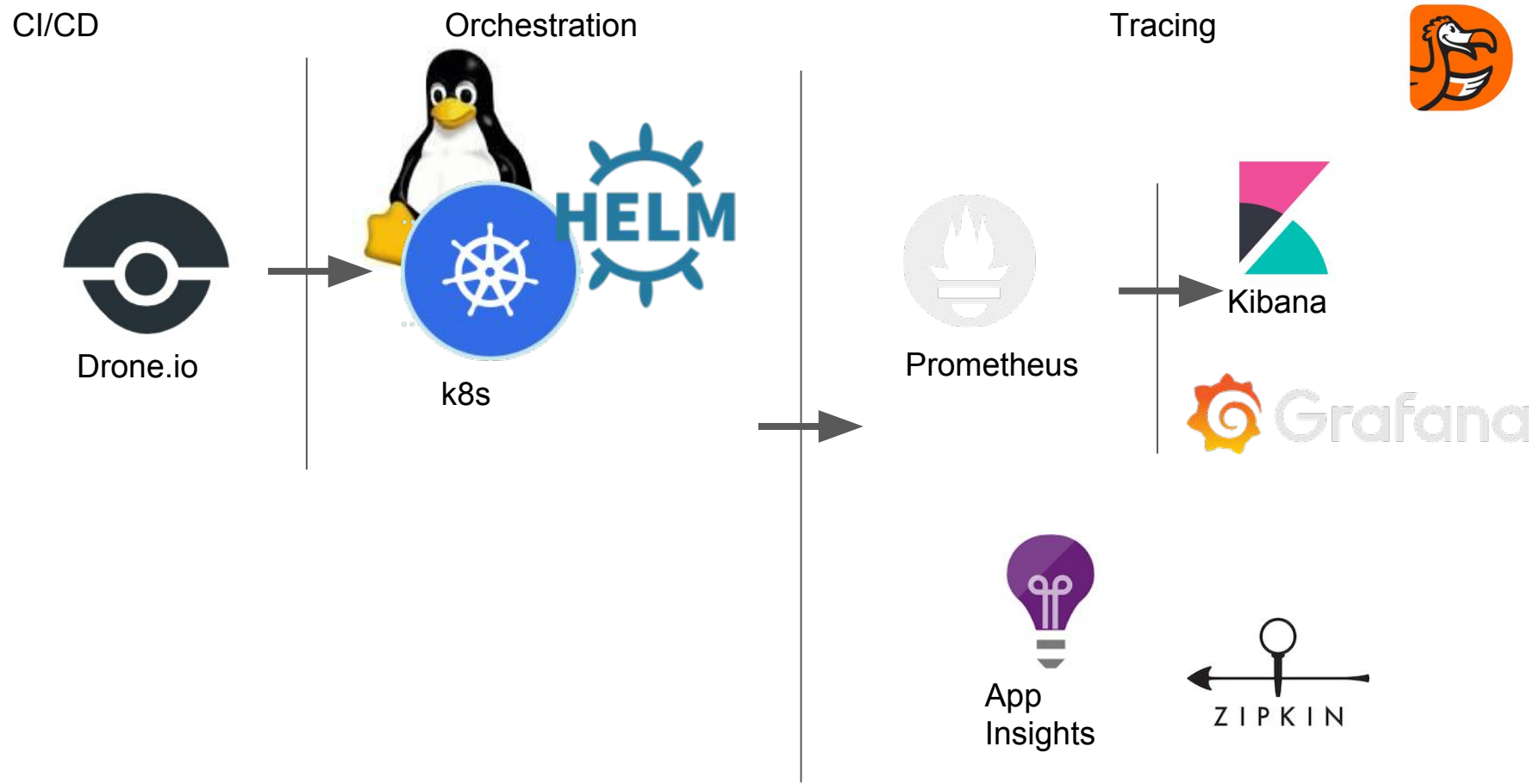
ASP.NET Core + Kubernetes

MySql

Redis

RabbitMQ

Kafka


CI/CD
Orchestration
Tracing
Drone.io
HELM
k8s
Prometheus
Kibana
Grafana
App
Insights
ZIPKIN


THIS IS FINE
Dodo Pizza Engineering © 2018
dodois.com

Architecture

What is common?

Photo by Bert Kaufmann / CC BY

Photo by Tom Hilton / CC BY

Performance is not about speed

Reliability

Fault Tolerance

Resilience

Performance Architecture - Evolution

Single Database

Replication

Autonomous Services

Autonomity is not so easy to cook

Eventual Consistency

Local Queue pattern

Event Queues

Commutativity + Idempotence

Failure scenarios

Synchronous

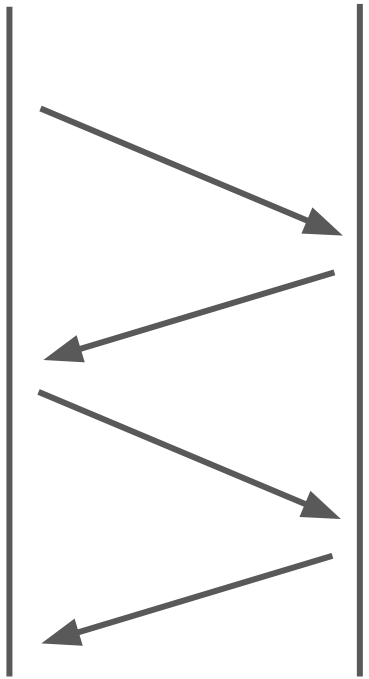
Asynchronous

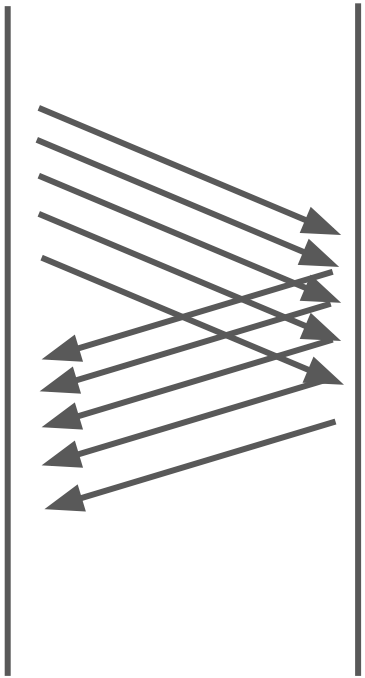
Retry induced

Cascading

Cold Start

Synchronous failures





How bad is multithreading?

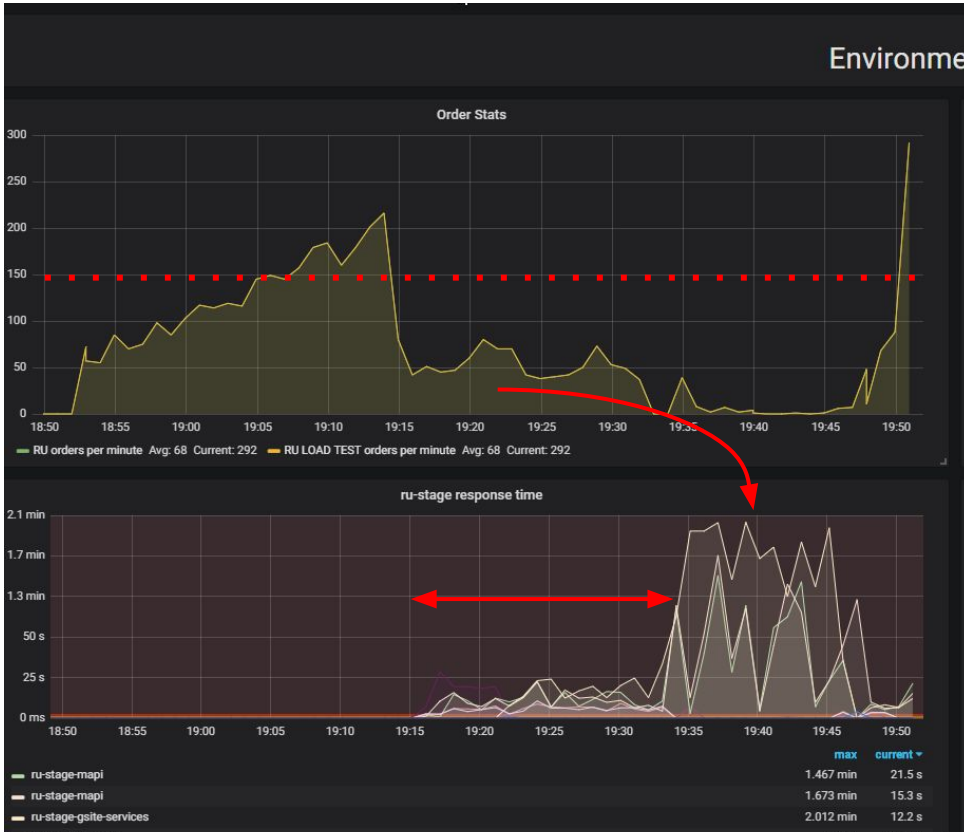
Awaken - Work - Sleep - Sleep- Sleep - Sleep - Awaken-
Work - Sleep - Sleep- Sleep - Sleep -Awaken- Work -
Sleep - Sleep- Sleep - Awaken- Work - Sleep - Sleep-
Sleep - Sleep - Sleep - Sleep - Awaken- Work - Sleep -
Sleep- Sleep - Sleep - Awaken- Work - Sleep - Sleep-
Sleep - Sleep -Awaken- Work - Sleep - Sleep- Sleep -
Sleep -Awaken- Work - Sleep - Sleep- Sleep - Sleep
-Awaken- Work - Sleep - Sleep- Sleep - Sleep -Awaken-
Work - Sleep - Sleep- Sleep - Sleep -Awaken- Work -

Small tasks are inefficient

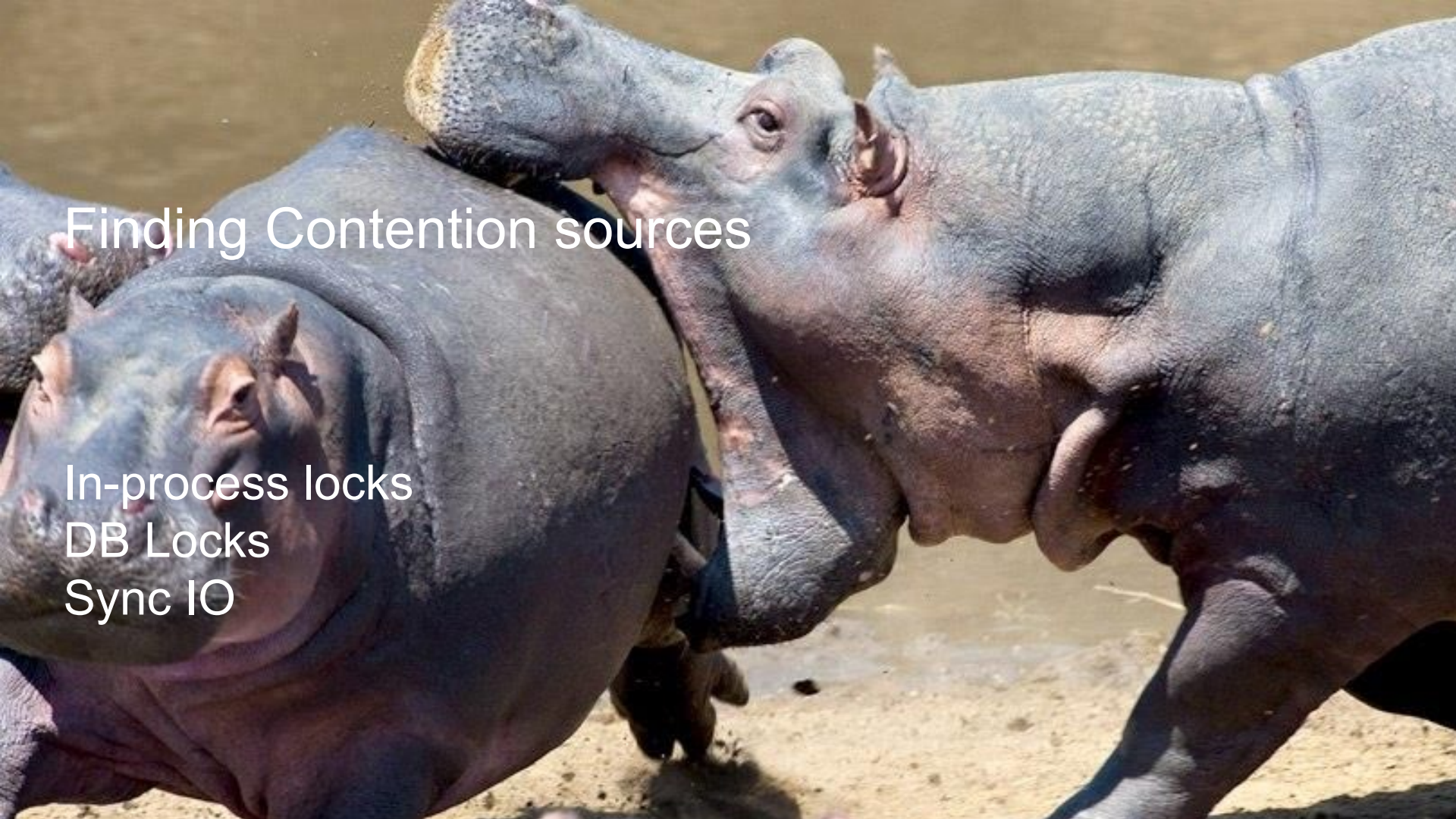
Duration (ns)	Efficiency (threads = CPUs)	Efficiency (threads = CPUs * 4)	OPS
911,773062	0,236112166	0,004880132055	1096764,142
1072,946908	0,2604661196	0,005507242104	932012,5649
1287,53447	0,3343987677	0,008724939086	776678,2351
1547,229543	0,3775715227	0,009172368868	646316,5112
1854,761383	0,4105167502	0,02030898985	539152,9116
2230,496819	0,4980597983	0,02062288907	448330,6103
2673,779667	0,6371534847	0,03455194077	374002,3953
3200,12857	0,6088520931	0,04107908932	312487,4448
3855,629573	0,7628318239	0,4247843257	259361,0151
4663,917168	0,7349478199	0,4715555852	214412,0412
5580,791322	0,8806853938	0,5752980095	179186,0584
6784,129448	0,7433827991	0,3342572692	147402,8477
7998,262302	0,6611580014	0,7700892508	125027,1574
9593,038097	0,7742571389	0,7241112544	104242,263
11493,25957	0,7954720566	0,743379935	87007,51897
13779,76998	0,8715313056	0,7647983214	72570,15184
16539,01032	0,5620537613	0,716203919	60463,10999
19906,25283	0,5931893267	0,6314297738	50235,47167

5580,791322	0,8806853938	0,5752980095	179186,0584
6784,129448	0,7433827991	0,3342572692	147402,8477
7998,262302	0,6611580014	0,7700892508	125027,1574
9593,038097	0,7742571389	0,7241112544	104242,263
11493,25957	0,7954720566	0,743379935	87007,51897
13779,76998	0,8715313056	0,7647983214	72570,15184
16539,01032	0,5620537613	0,716203919	60463,10999
19906,25283	0,5931893267	0,6314297738	50235,47167
23976,55761	0,8714963311	0,7421941946	41707,40506
28583,09946	0,7969763186	0,8107313003	34985,709
34464,63605	0,7956718594	0,7231070067	29015,24909
41304,57708	0,824060509	0,7963758869	24210,39194
49755,22062	0,7556066422	0,7397020608	20098,39345
59298,68819	0,8268080369	0,7490853605	16863,77946
71081,95955	0,711466309	0,7544574978	14068,2672
85324,48769	0,8277507232	0,8129600316	11719,96489
102525,0034	0,7684888271	0,759371057	9753,718278
123087,3228	0,8494819015	0,7279135349	8124,313515
151536,6568	0,8141496776	0,7486169477	6599,063364
179039,7436	0,8526226458	0,921453582	5585,352056

Slow server death

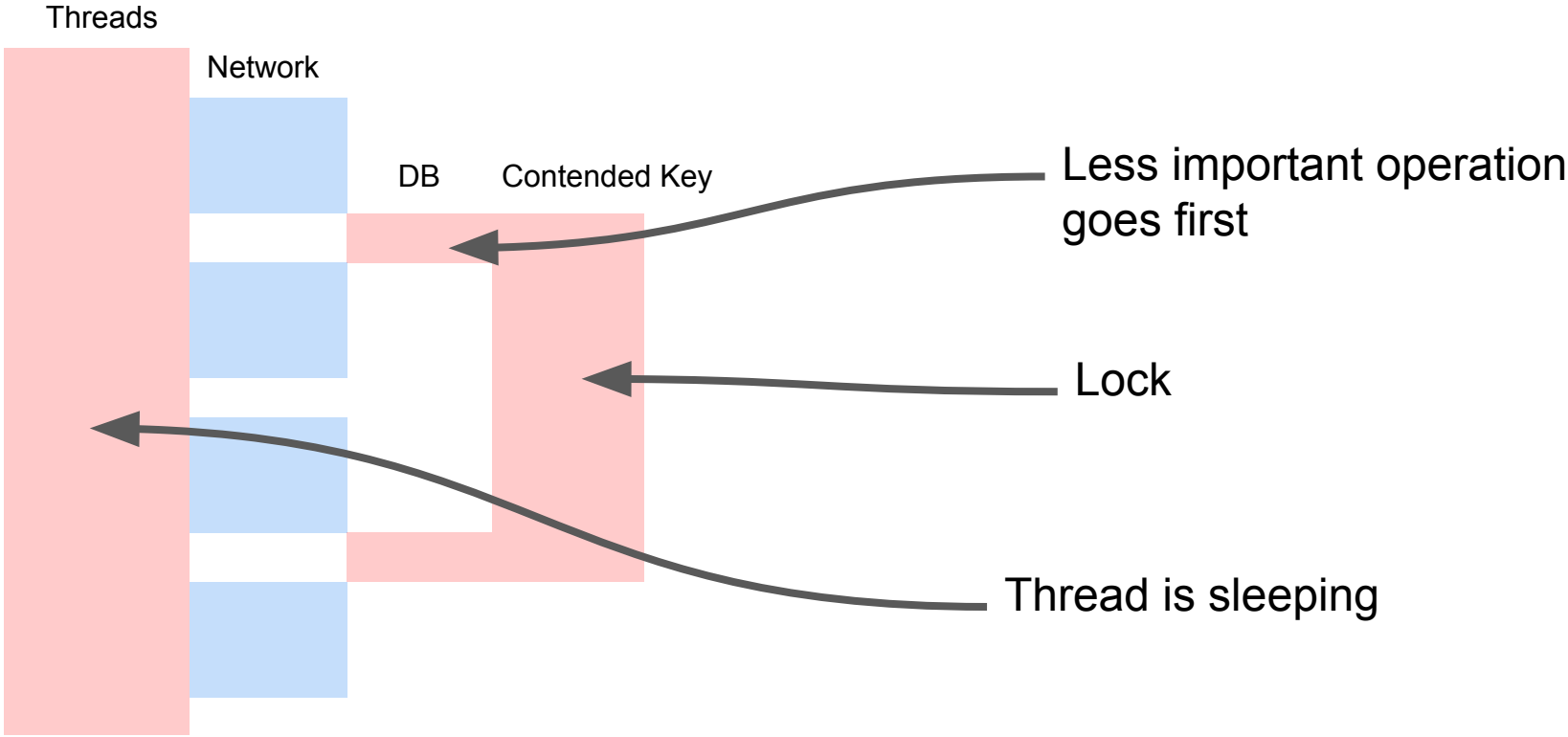


Finding Contention sources



In-process locks
DB Locks
Sync IO

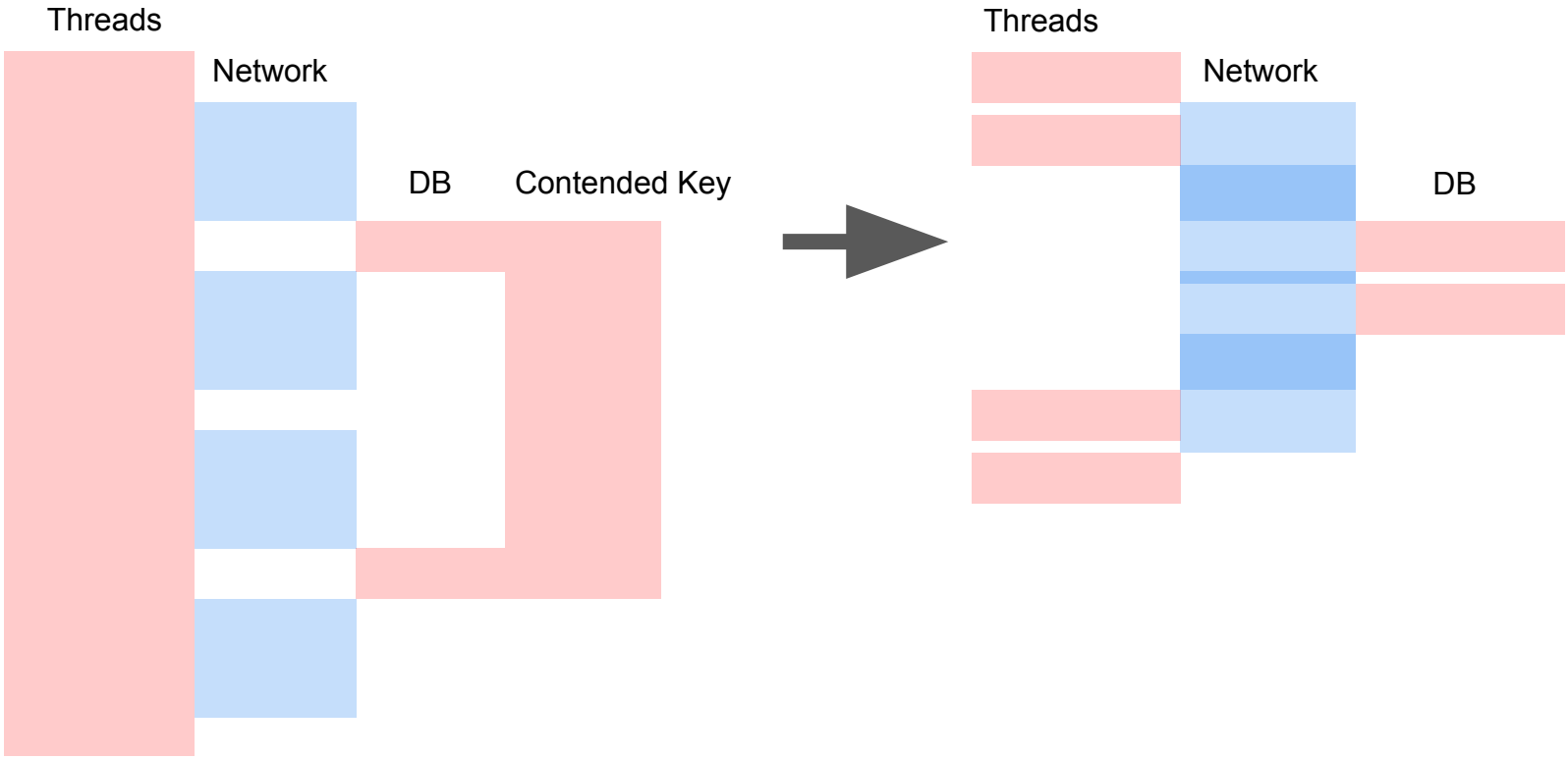
What’s wrong with blocking code?



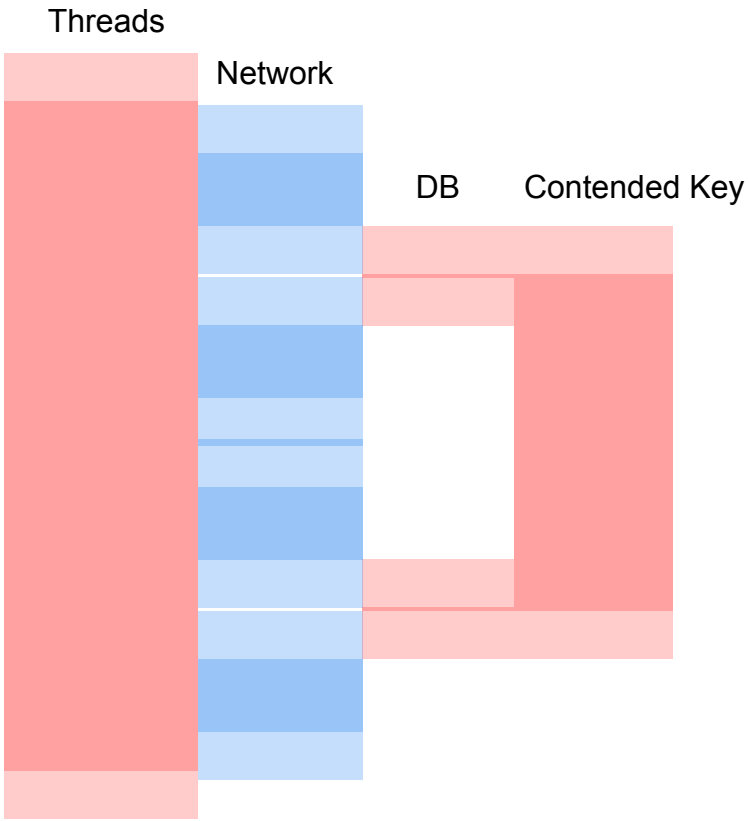
Eventual Consistency for secondary operation

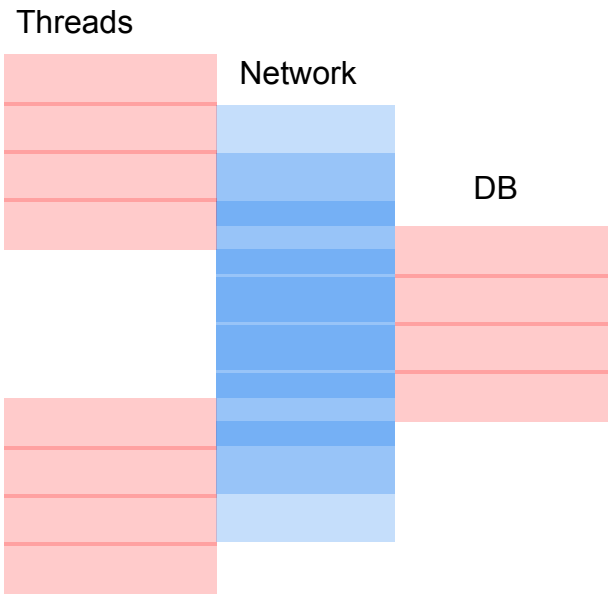
```
var fallback = FallbackPolicy<OptionalData>
   .Handle<OperationCancelledException>()
   .FallbackAsync<OptionalData>(OptionalData.Default);

var optionalDataTask = fallback
   .ExecuteAsync(async () => await CalculateOptionalDataAsync());

//…

var required = await CalculateRequiredData();

var optional = await optionalDataTask;

var price = CalculatePriceAsync(optional, required);
```

TPL async + lighter consistency constraints

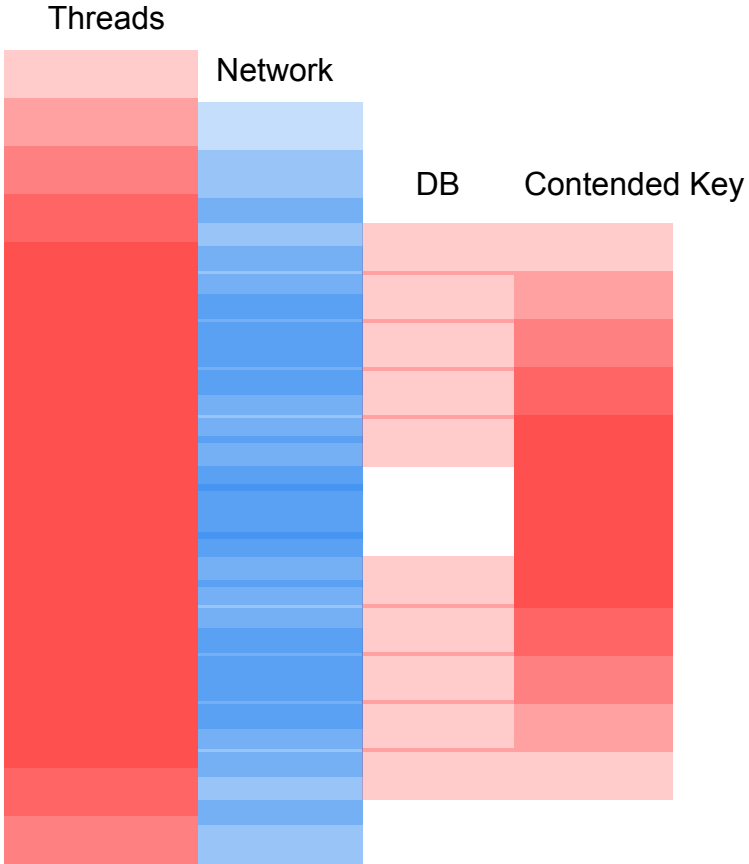


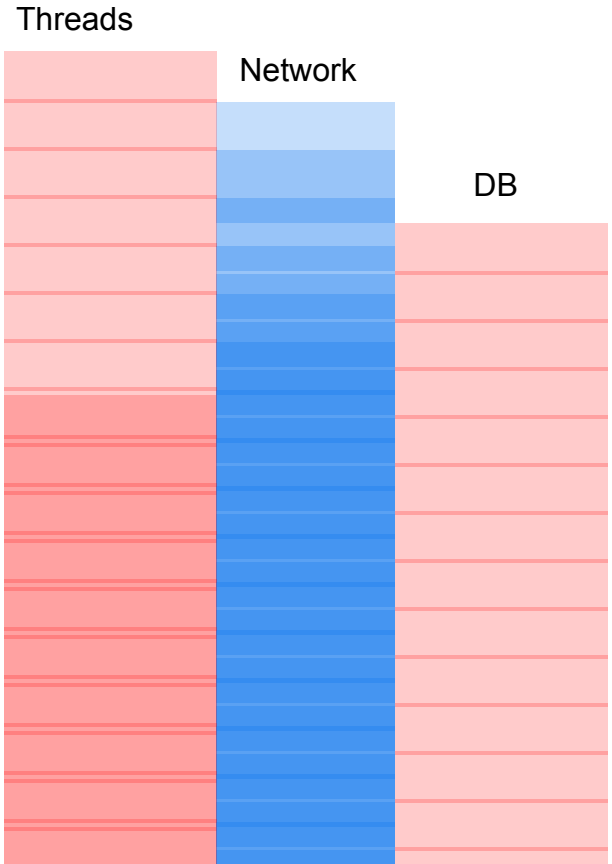
Which one gets saturated?





Which one can make progress?





Do you still want this file log?

Azure standard disk latency:

~5ms.

Rate:

~200 writes/s

https://blogs.technet.microsoft.com/andrewc/2016/09/09/understanding-azure-virtual-machine-iops-throughput-and-disk-latency/

NLog

2866	2867	2888	2889

2272 Threads (78% of all threads) have this same call stack.
Note: Grouping of identical threads can be disabled in the 'Preferences' tab of the Ana

.NET Call Stack

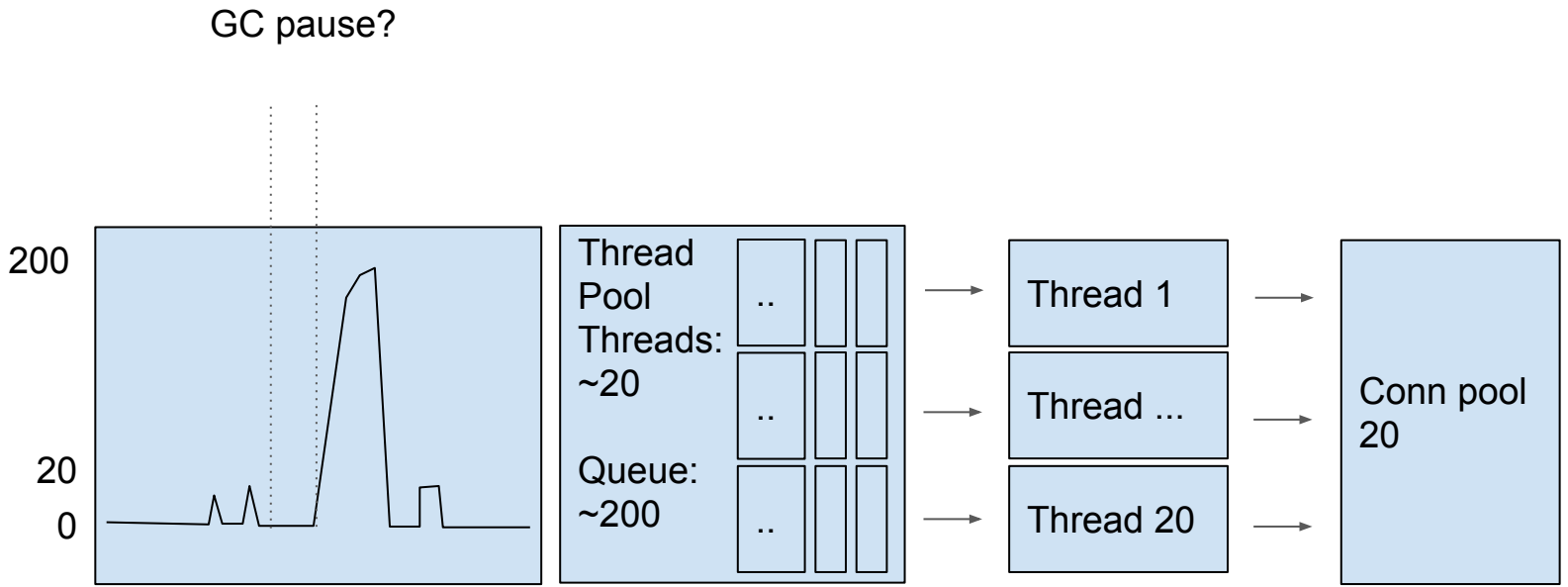
[[GCFrame]]
[[GCFrame]]
[[HelperMethodFrame] (System.Threading.Monitor.Enter)] System.Threading.Monitor.Enter(System.Object)
NLog.Targets.Target.WriteAsyncThreadSafe(NLog.Common.AsyncLogEventInfo)+51
NLog.Targets.Target.WriteAsyncLogEvent(NLog.Common.AsyncLogEventInfo)+107
NLog.LoggerImpl.WriteToTargetWithFilterChain(NLog.Targets.Target, NLog.Filters.FilterResult, NLog.LogEve

Fix NLog configuration with AsyncWrapper

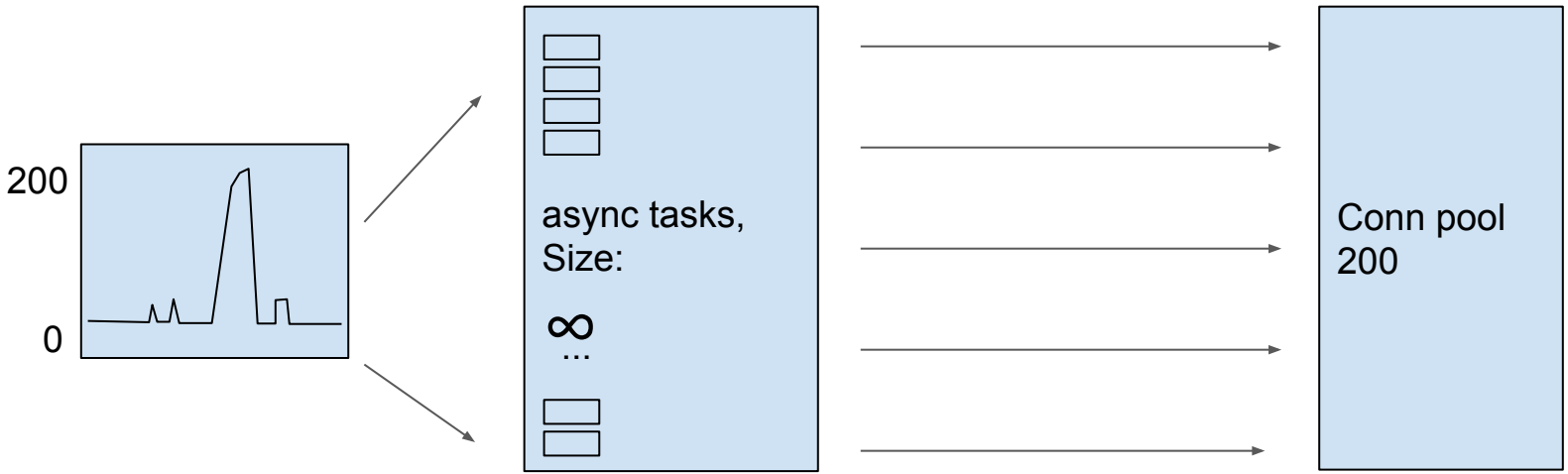
```
<targets>
   <target xsi:type="File" name="file" fileName="C:\logs\${shortdate}.log"
           layout="${longdate} ${logger} ${uppercase:${level}} ${message}" />

   <wrapper-target xsi:type="AsyncWrapper" name="asyncFile" batchSize="1000" >
     <target-ref name="file"/>
   </wrapper-target>
 </targets>

 <rules>
   <logger name="async" minlevel="Trace" writeTo="asyncFile" />
   <logger name="blocking" minlevel="Trace" writeTo="file" />
 </rules>
```

Asyncronous failures

Thread Pool regulates concurrency



Async restores concurrency



∞ VS... DB

∞

VS...

Inside DB:
- CPU
- memory
- network
- storage

async VS... limited resources

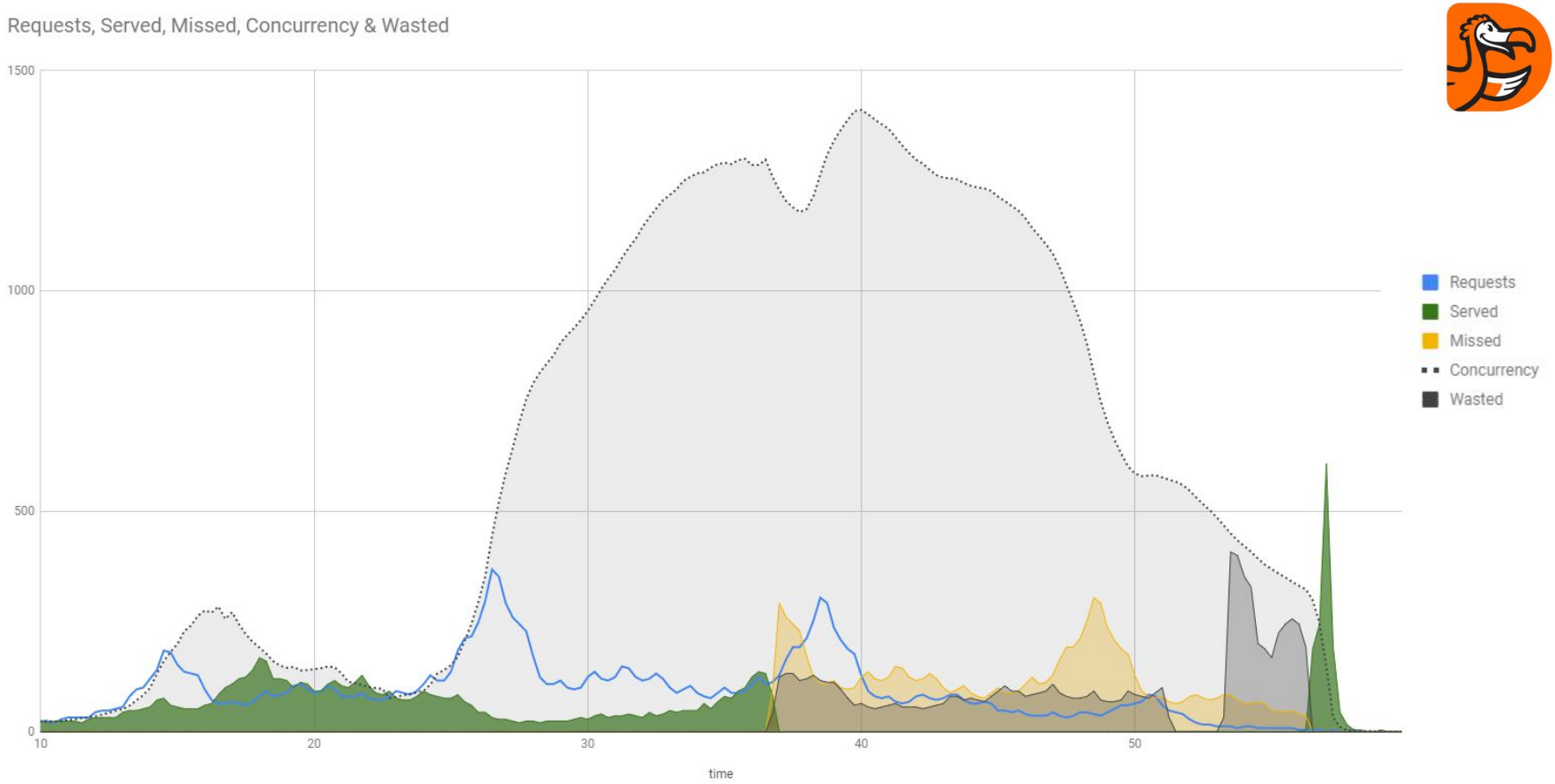
concurrency VS... parallelism

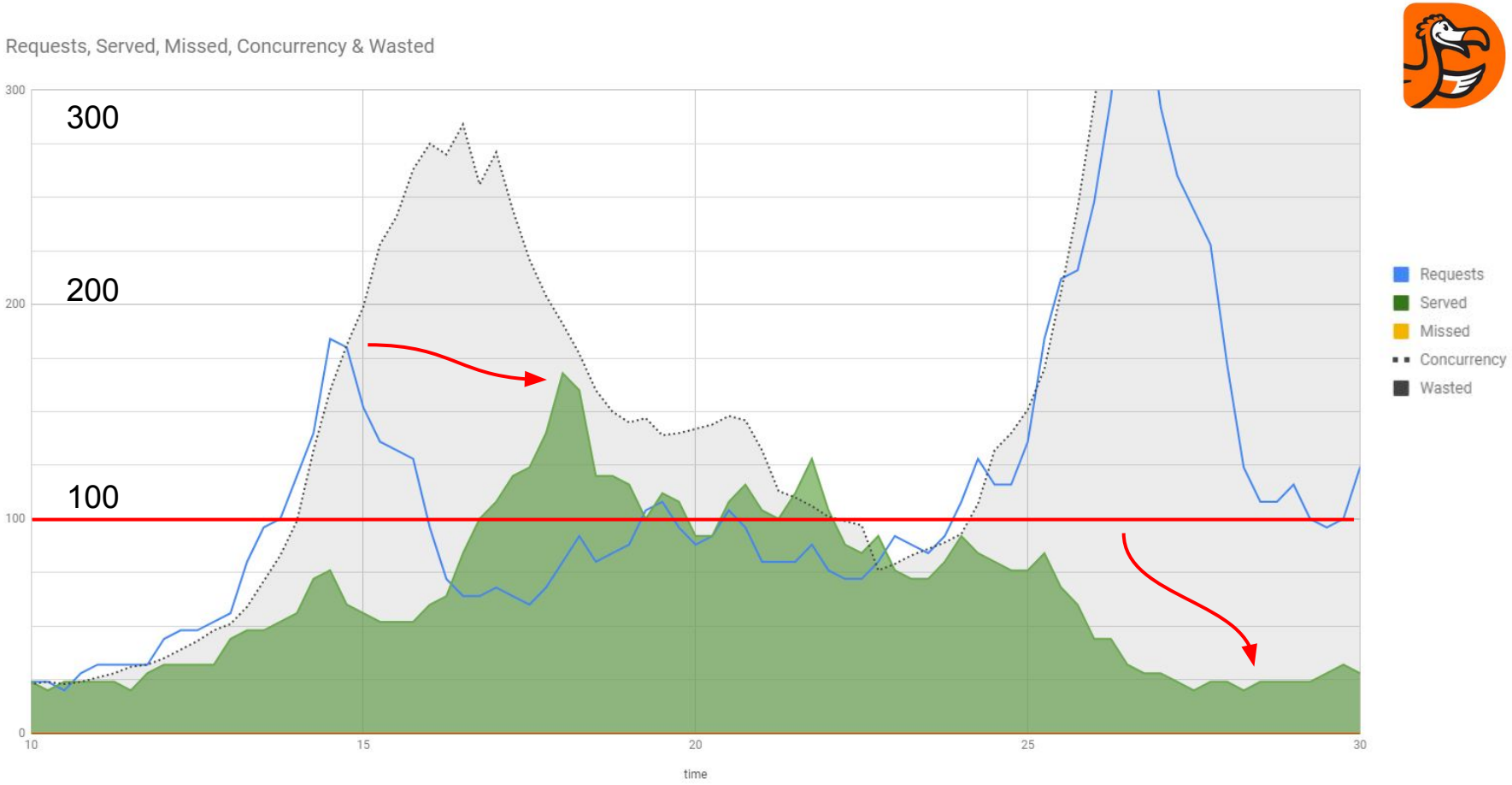
Concurrency is not parallelism

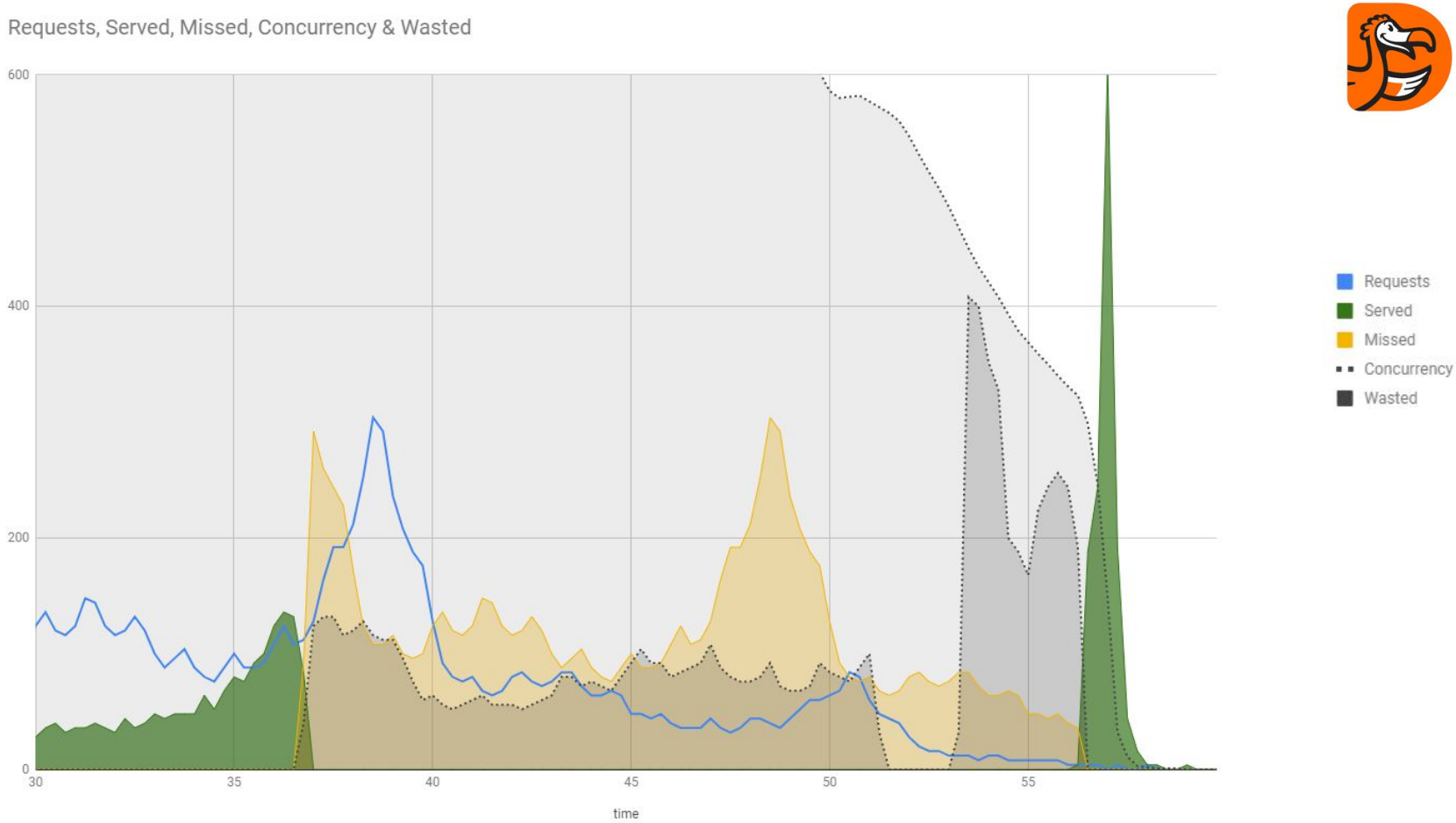
“Concurrency is about dealing with multiple things at the same time, parallelism is about doing multiple things at the same time”

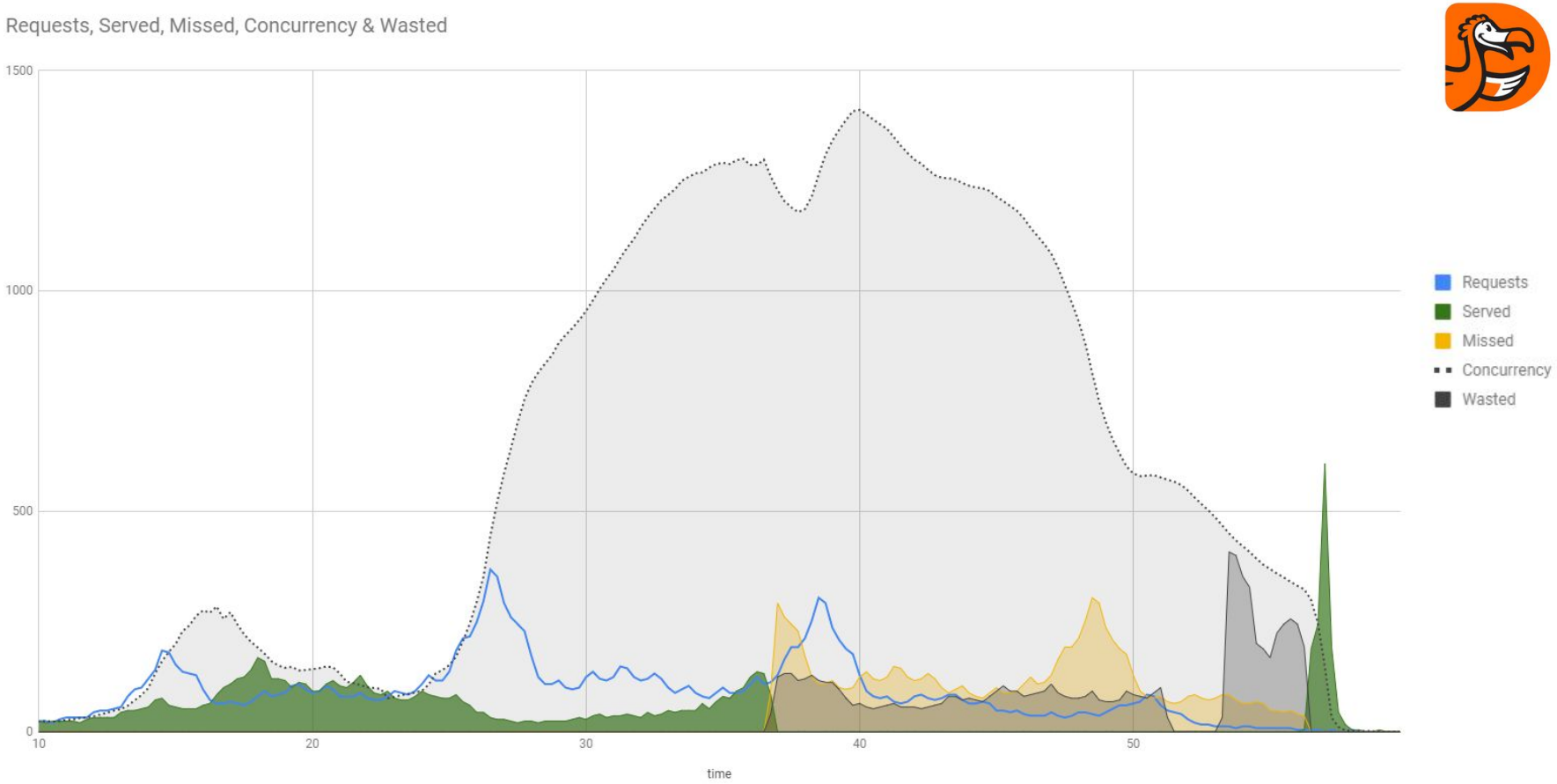
Rob Pike

https://blog.golang.org/concurrency-is-not-parallelism


Requests, Served, Missed, Concurrency & Wasted
1500
1000
500
0
10
20
30
40
50
time
Requests
Served
Missed
Concurrency
Wasted


Requests, Served, Missed, Concurrency & Wasted
300
200
100
0
10
15
20
25
30
time
Requests
Served
Missed
Concurrency
Wasted


Requests, Served, Missed, Concurrency & Wasted
600
400
200
0
30
35
40
45
50
55
time
Requests
Served
Missed
Concurrency
Wasted


Requests, Served, Missed, Concurrency & Wasted
1500
1000
500
0
10
20
30
40
50
time
Requests
Served
Missed
Concurrency
Wasted

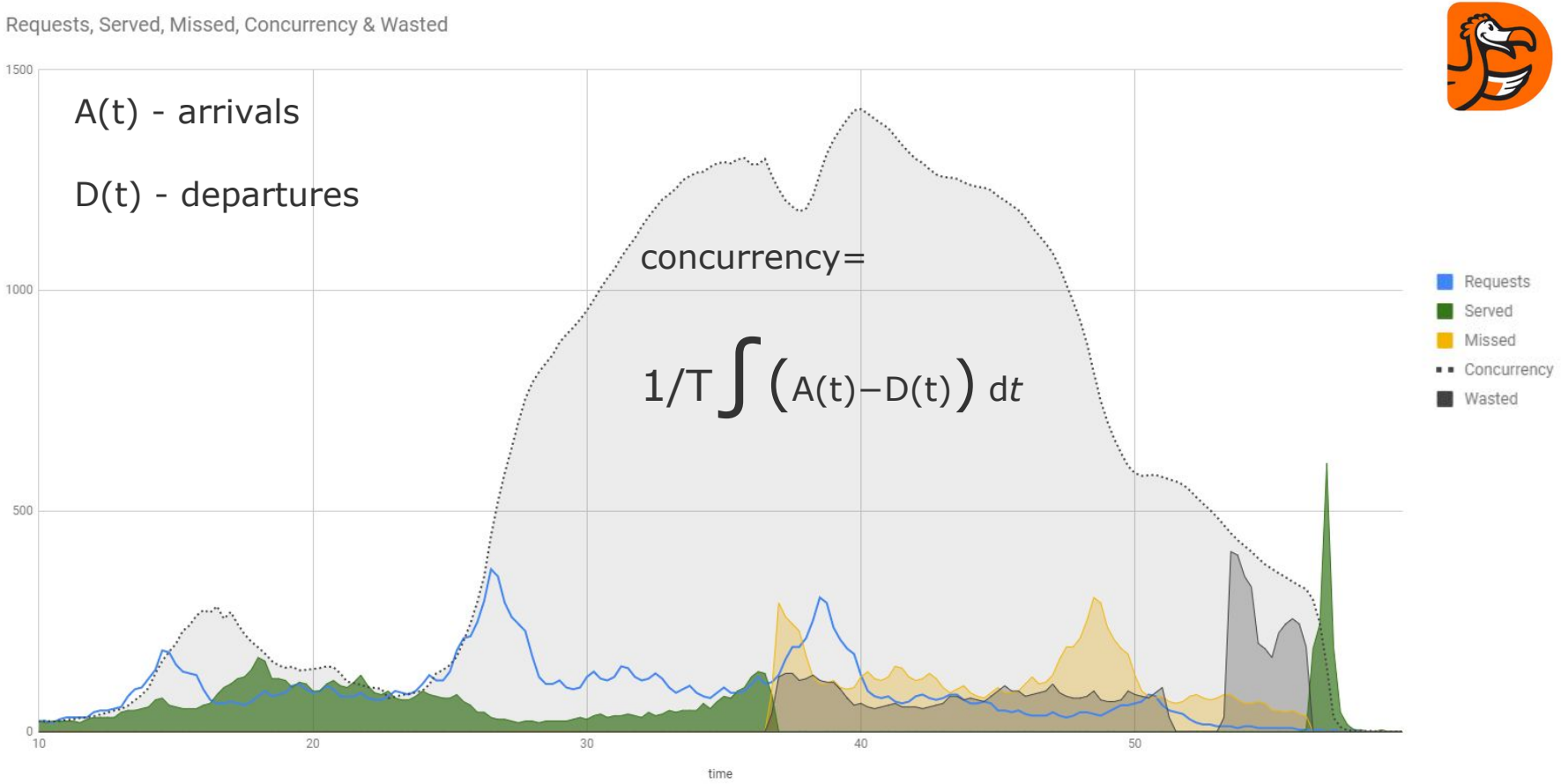
Little’s Law

$$L = \lambda W$$

L - number of customers inside

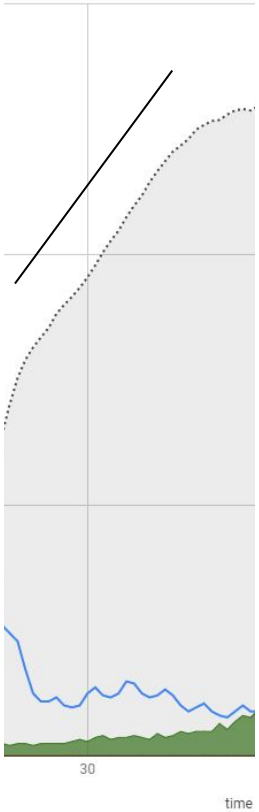
λ - arrival rate

W - time in the system


Requests, Served, Missed, Concurrency & Wasted
A(t) - arrivals
D(t) - departures
concurrency=
$1/T \int (A(t)-D(t))\, dt$
1500
1000
500
0
10
20
30
40
50
time
Requests
Served
Missed
Concurrency
Wasted

A(t) - arrivals

D(t) - departures



concurrency=

$$1/T \int (A(t) - D(t))\, dt$$

$$A(t) > D(t) \geq 0$$

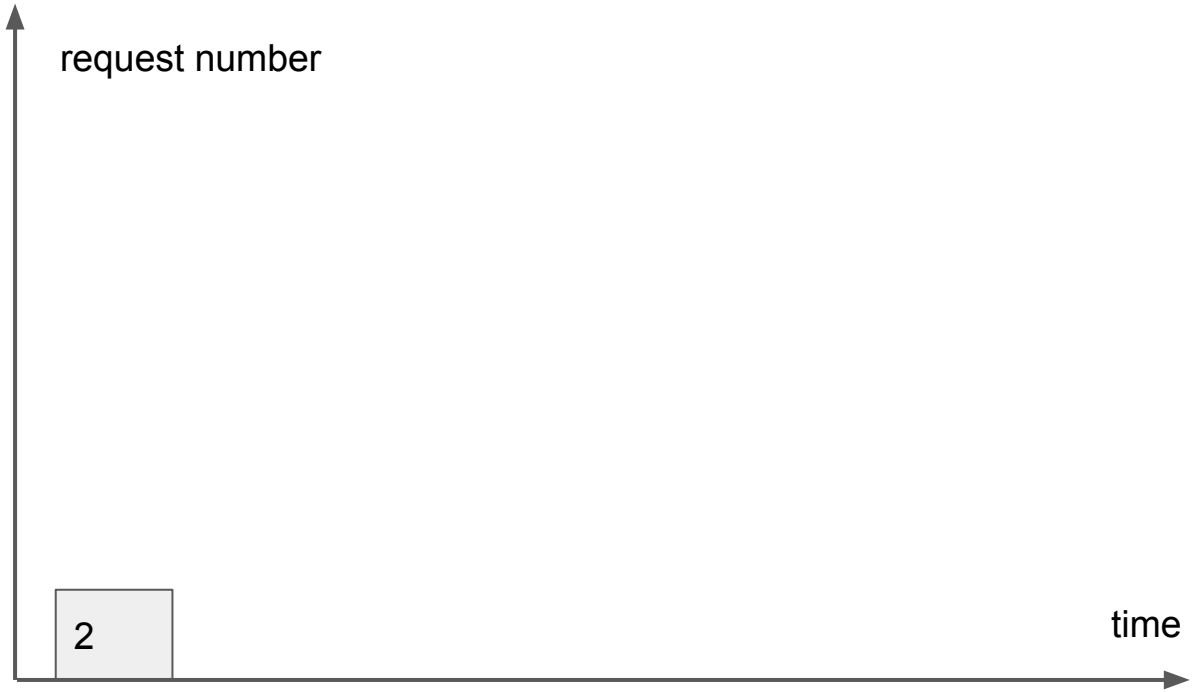
$$A(t) \approx C$$

Accumulating concurrency

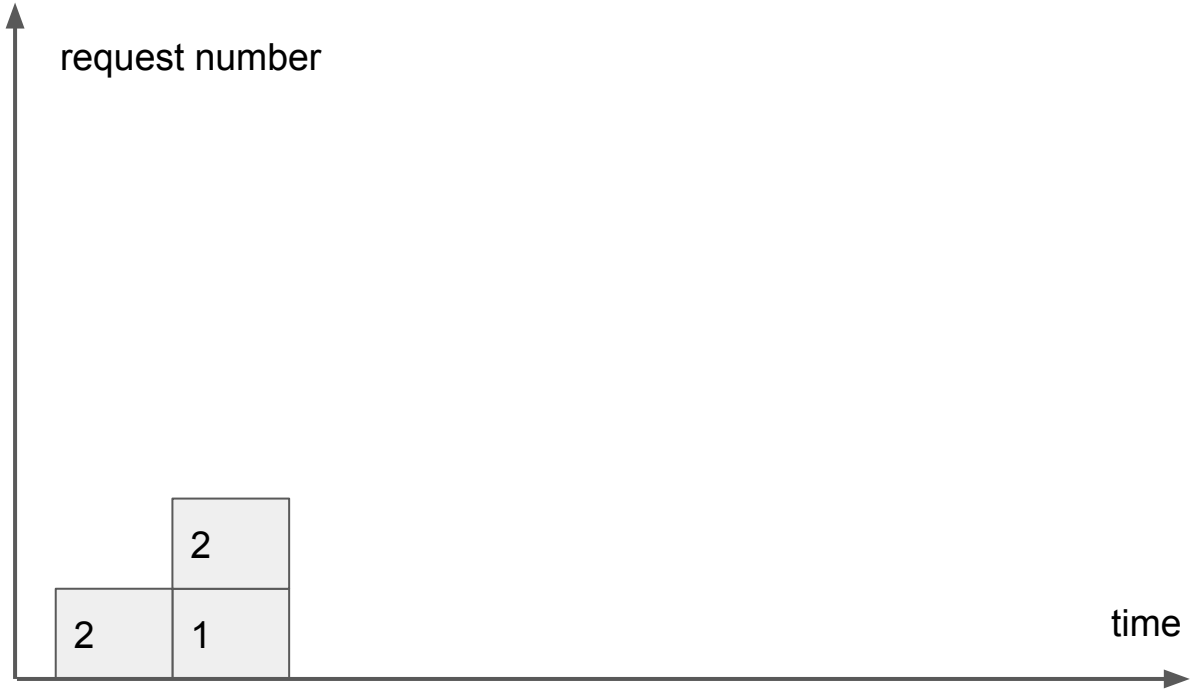
Game 1

Fair Scheduling

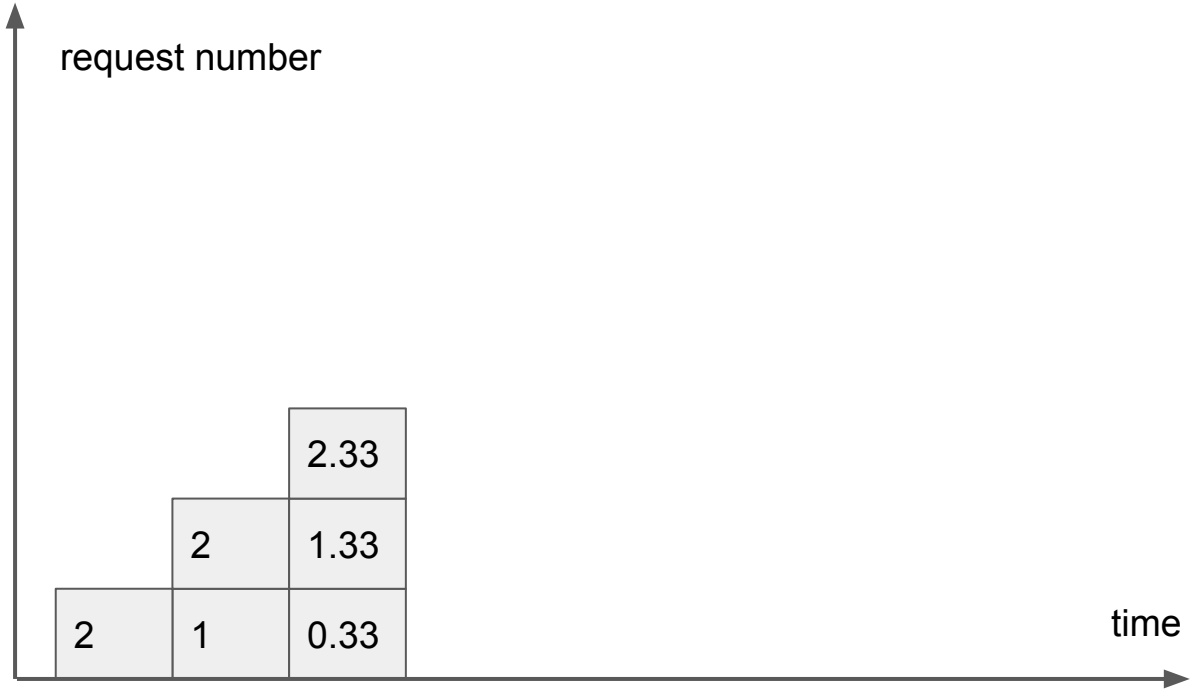
Accumulating concurrency



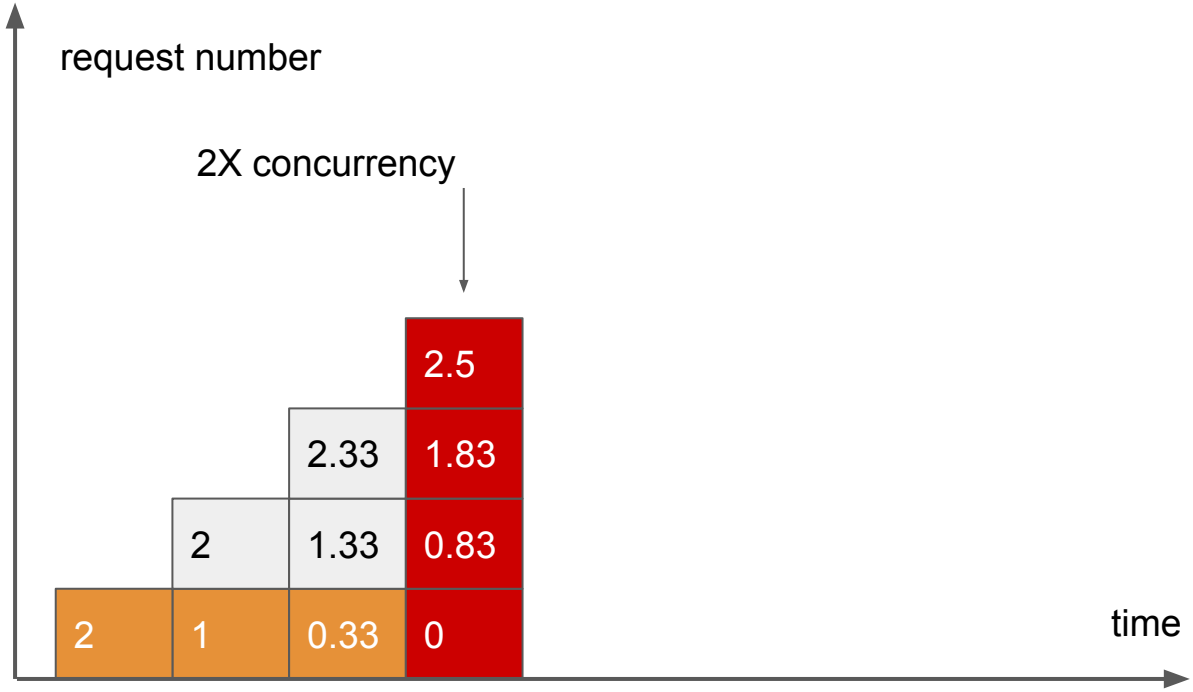
Accumulating concurrency



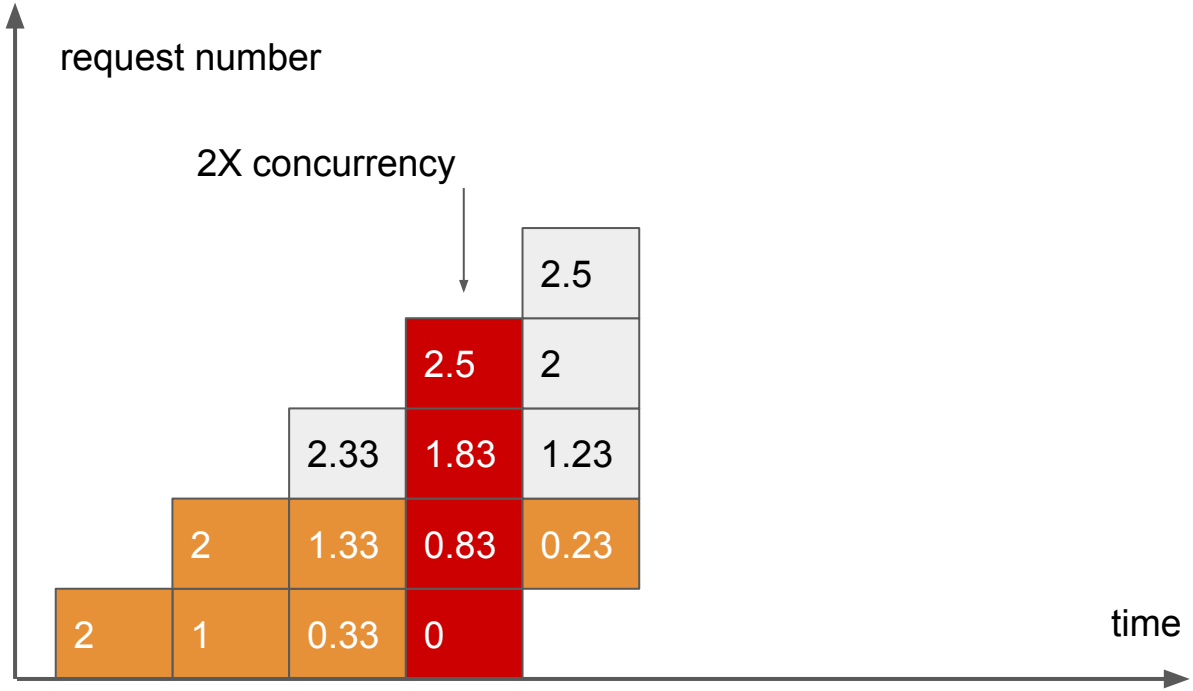
Accumulating concurrency



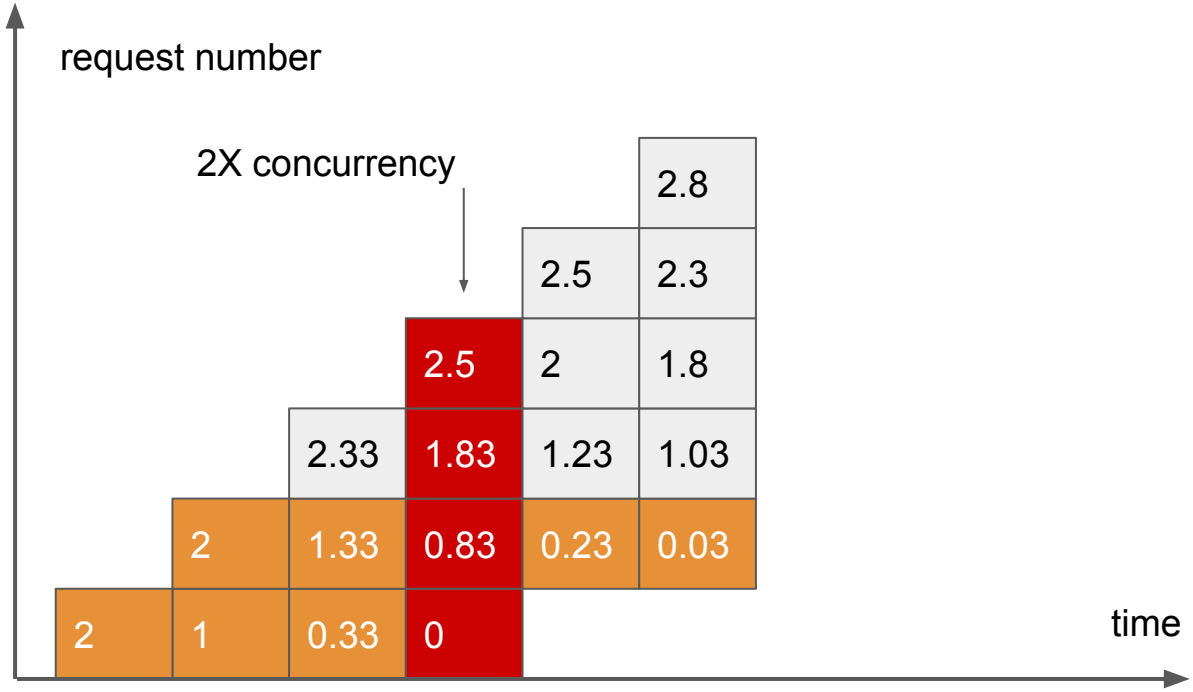
Accumulating concurrency



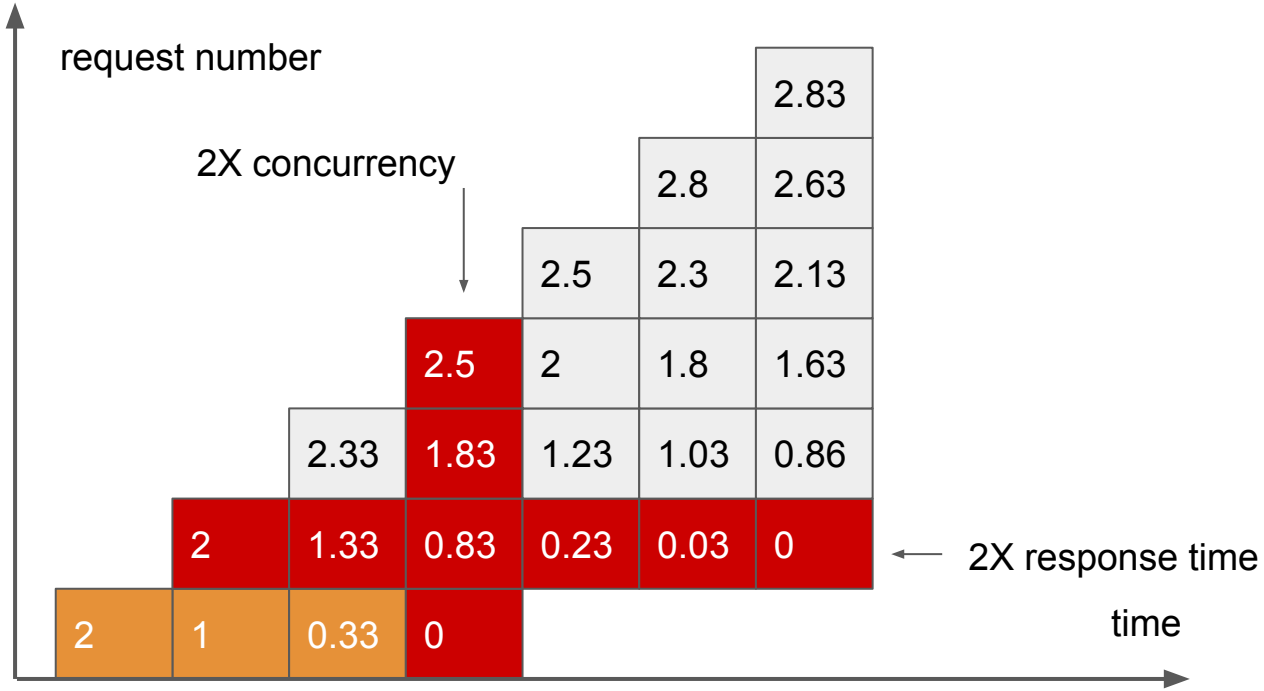
Accumulating concurrency



Accumulating concurrency



Accumulating concurrency



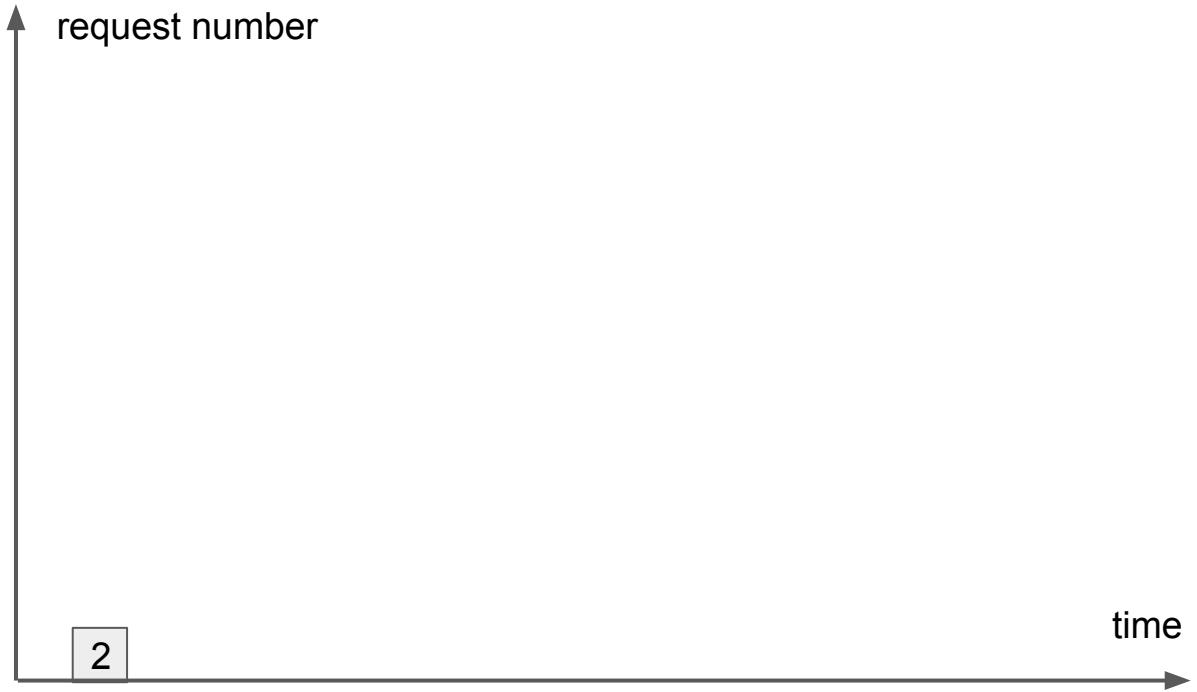
Controlled concurrency

Game 2

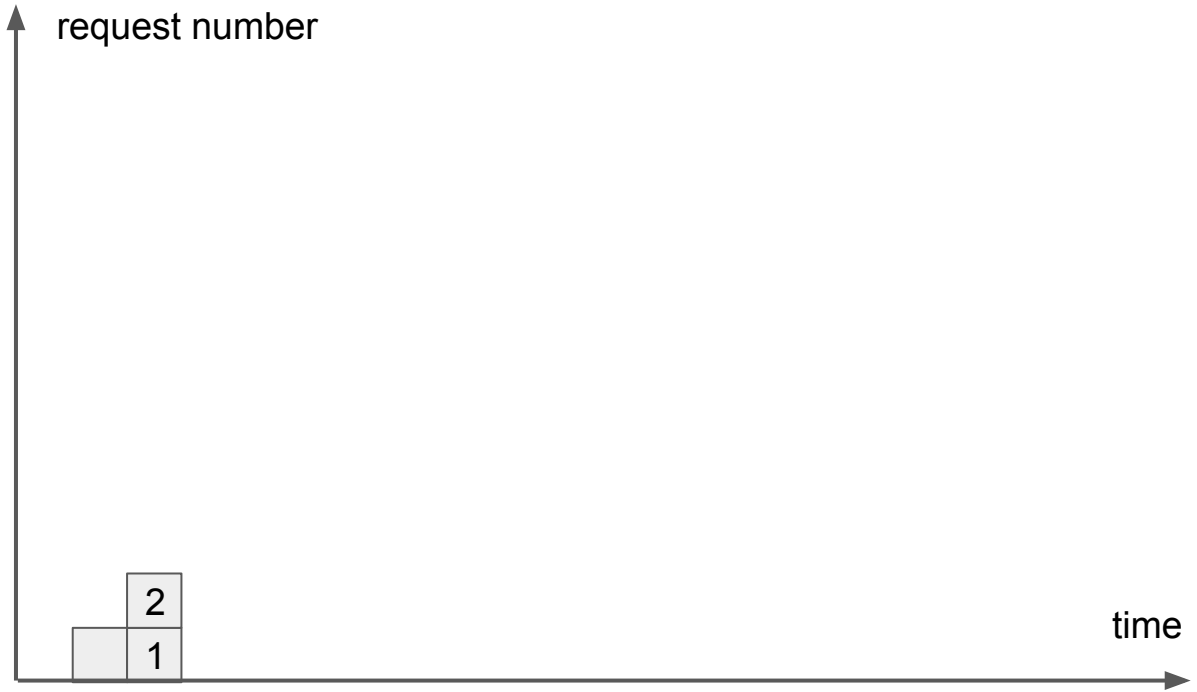
Concurrency Limit = 2

Queue Size = 2

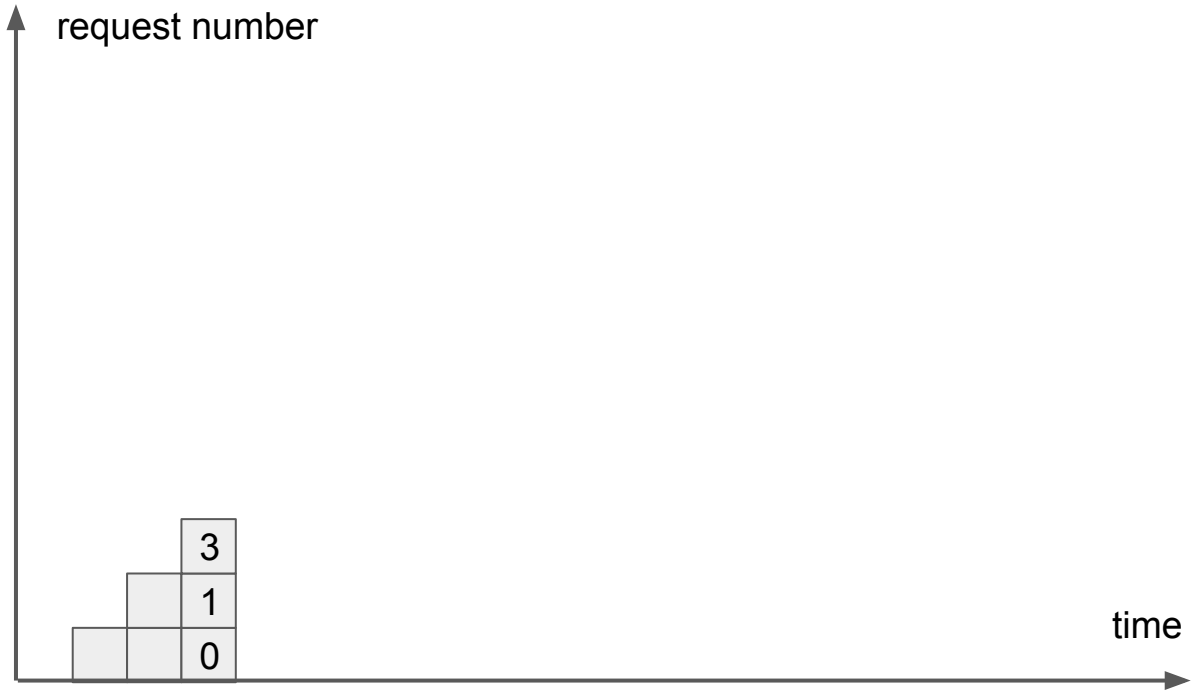
Controlled concurrency



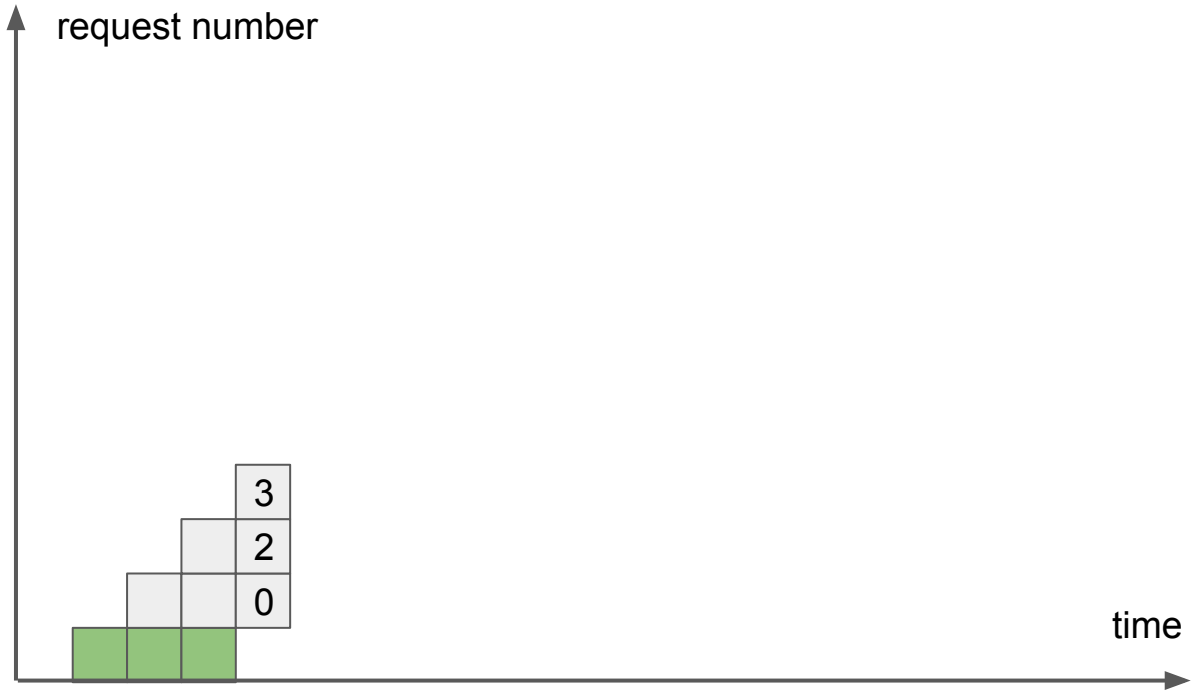
Controlled concurrency



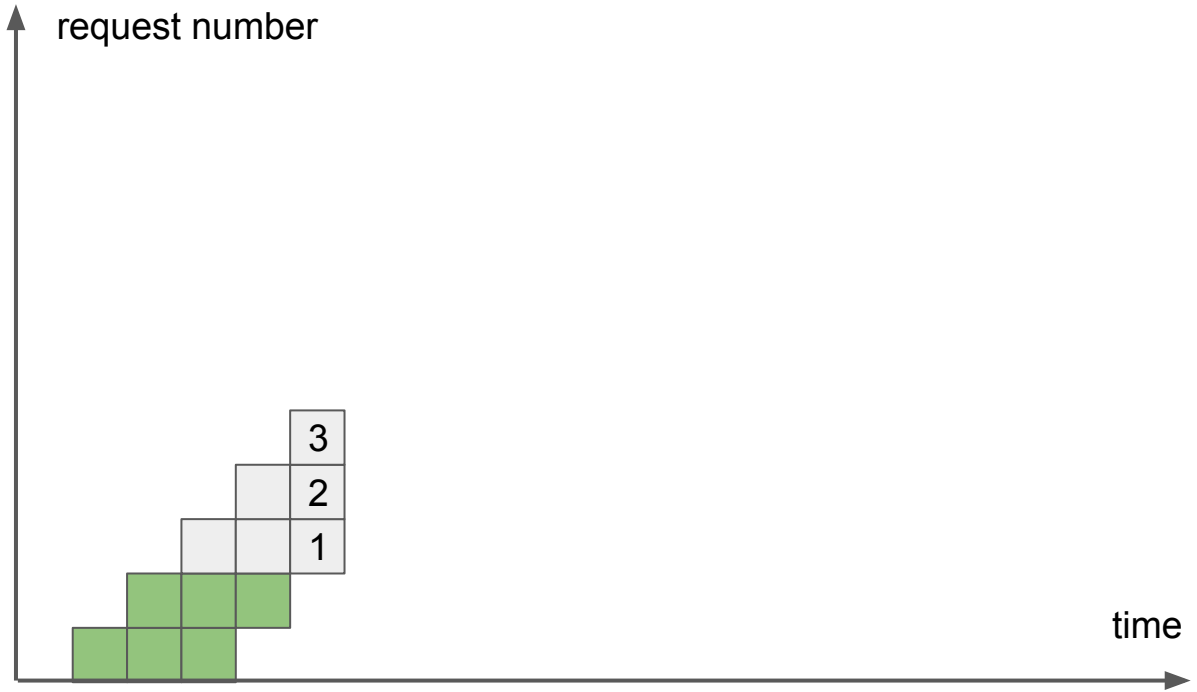
Controlled concurrency



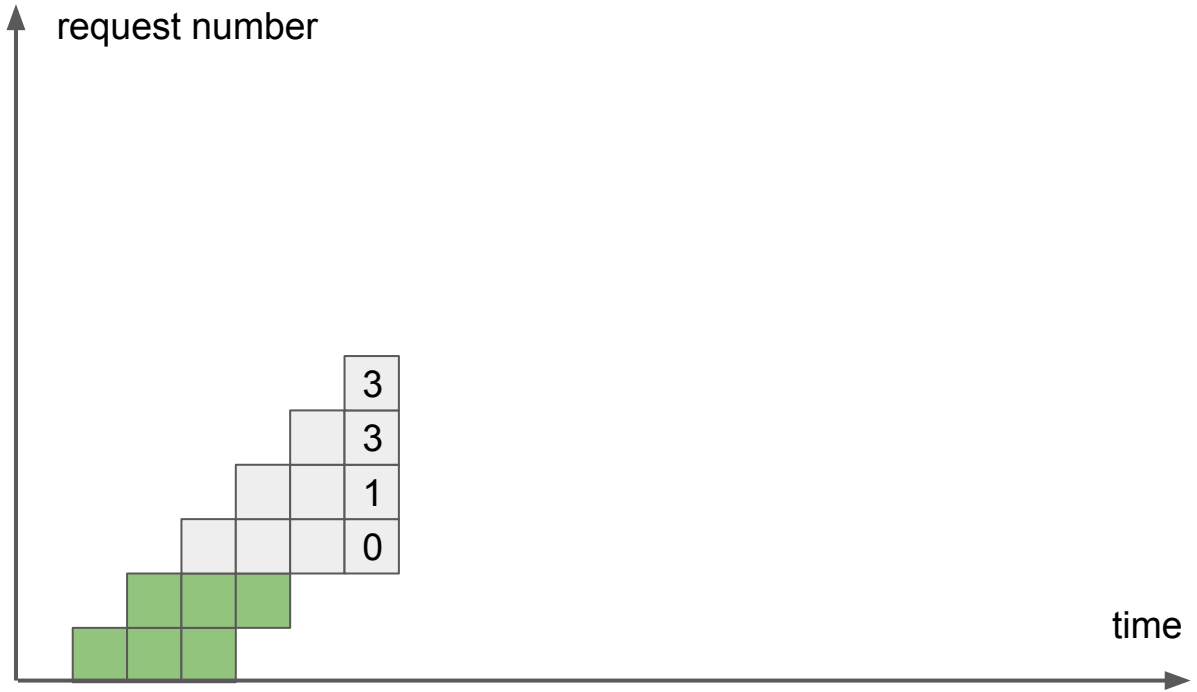
Controlled concurrency



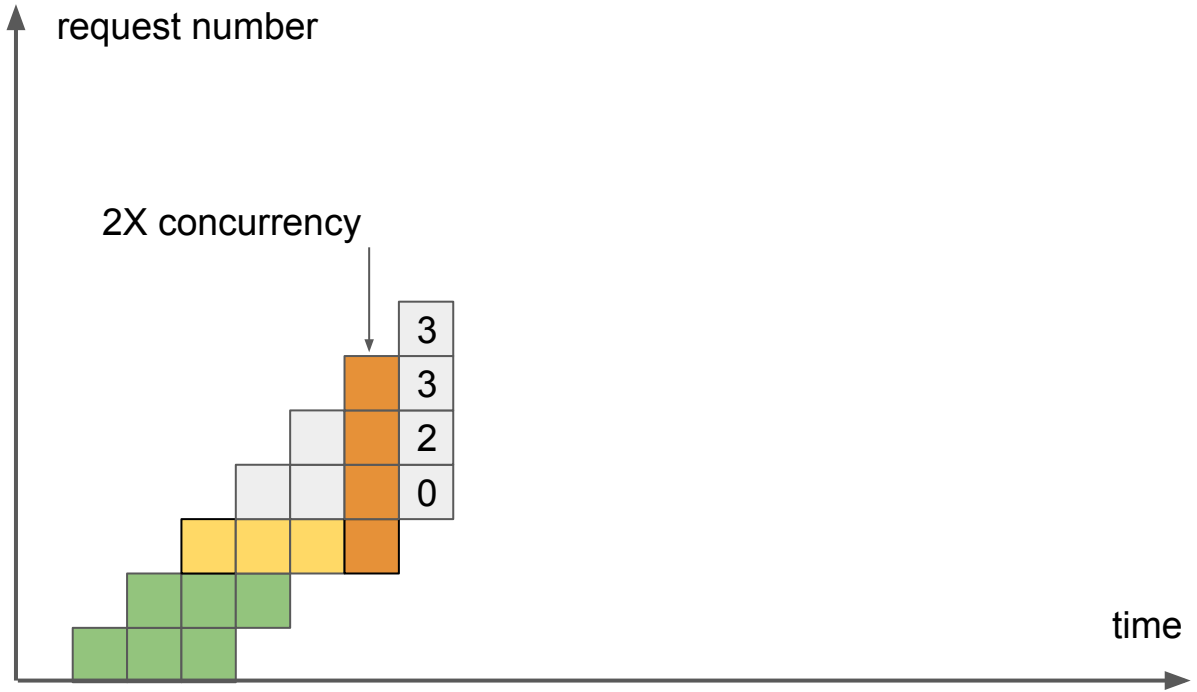
Controlled concurrency



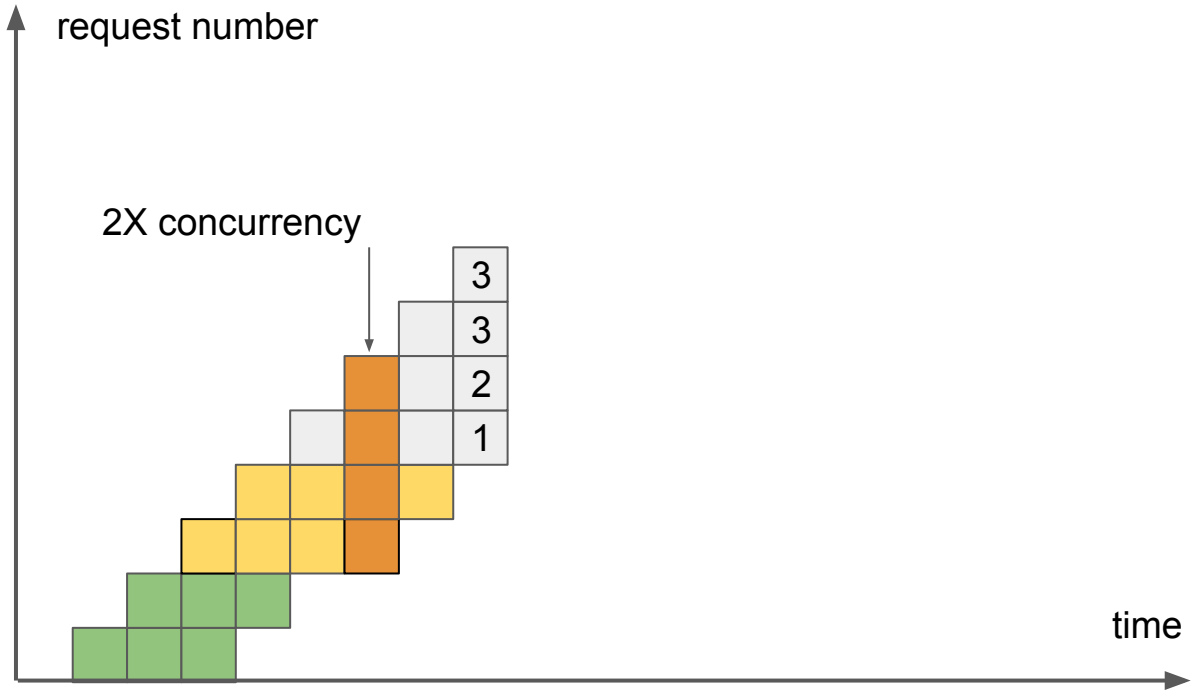
Controlled concurrency



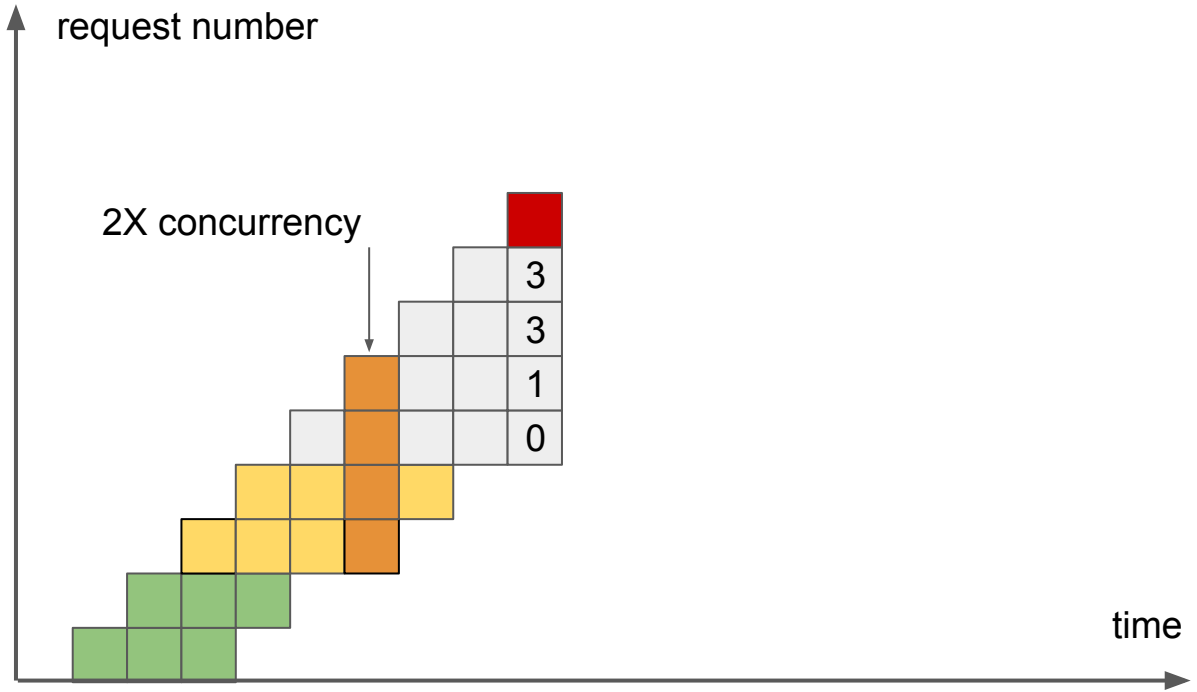
Controlled concurrency



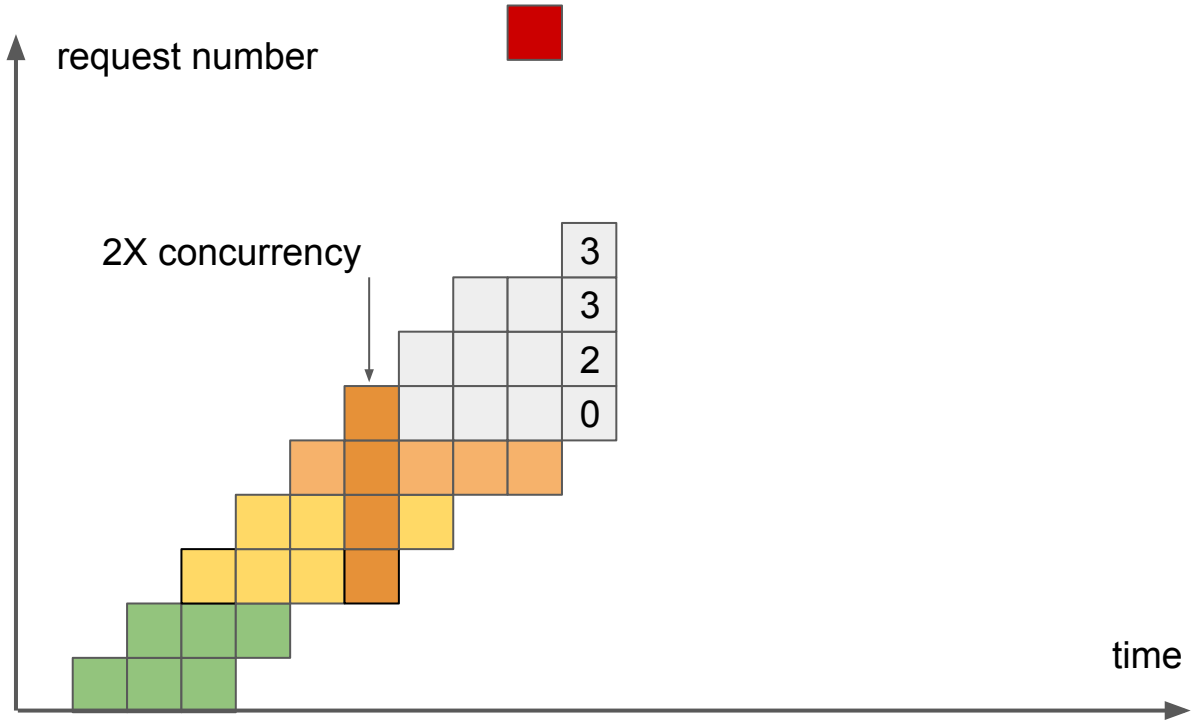
Controlled concurrency



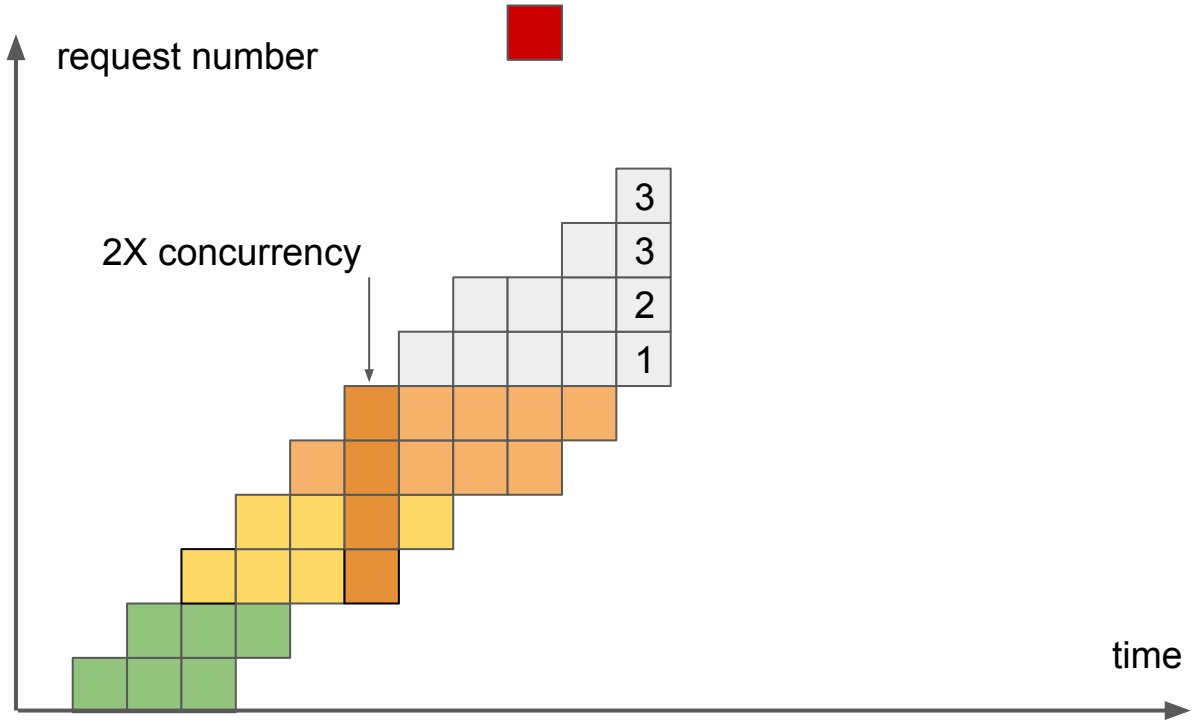
Controlled concurrency



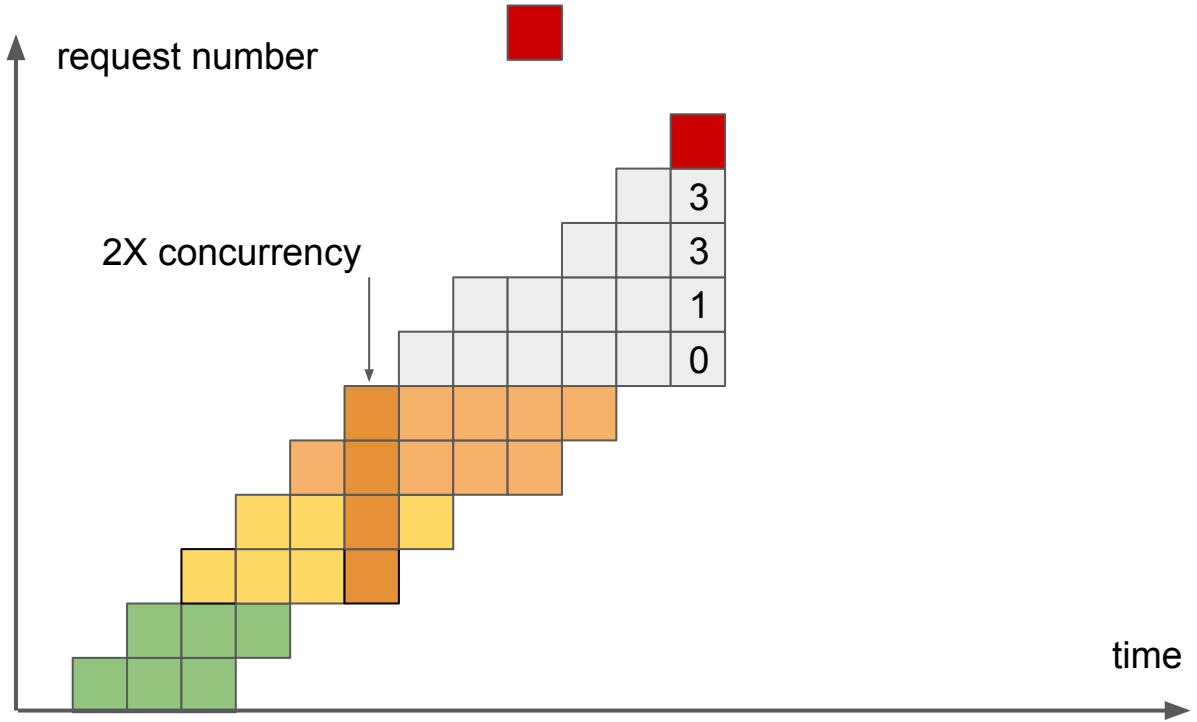
Controlled concurrency



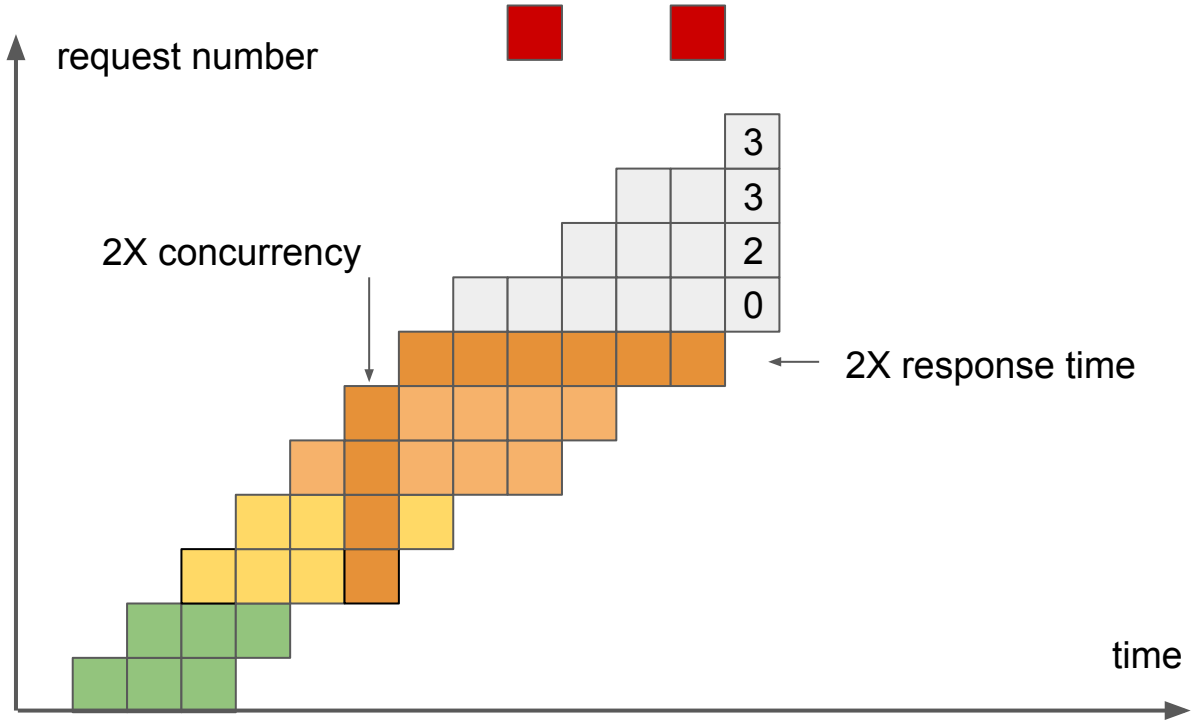
Controlled concurrency



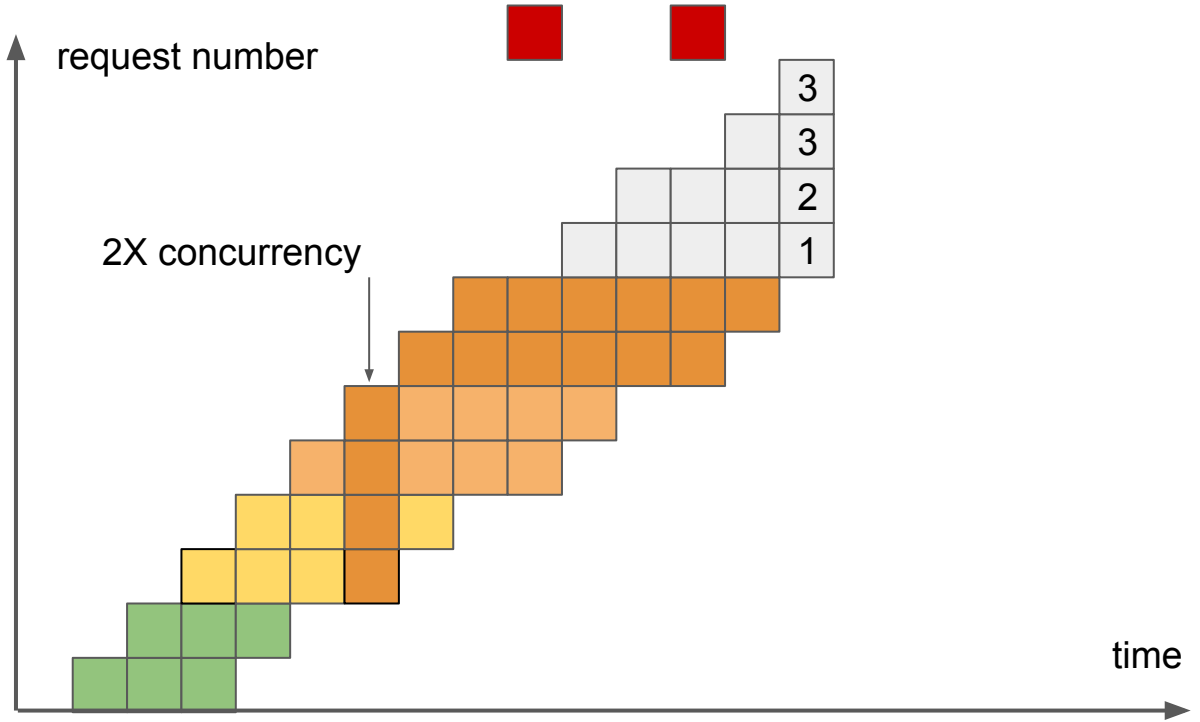
Controlled concurrency



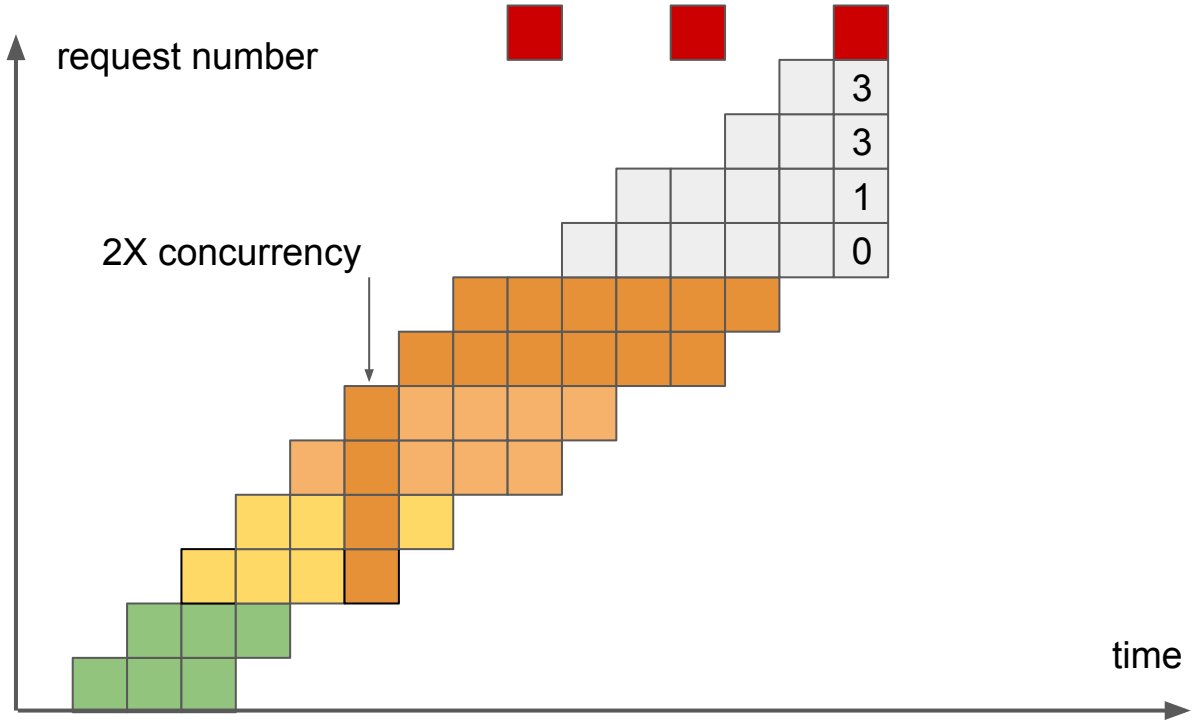
Controlled concurrency



Controlled concurrency

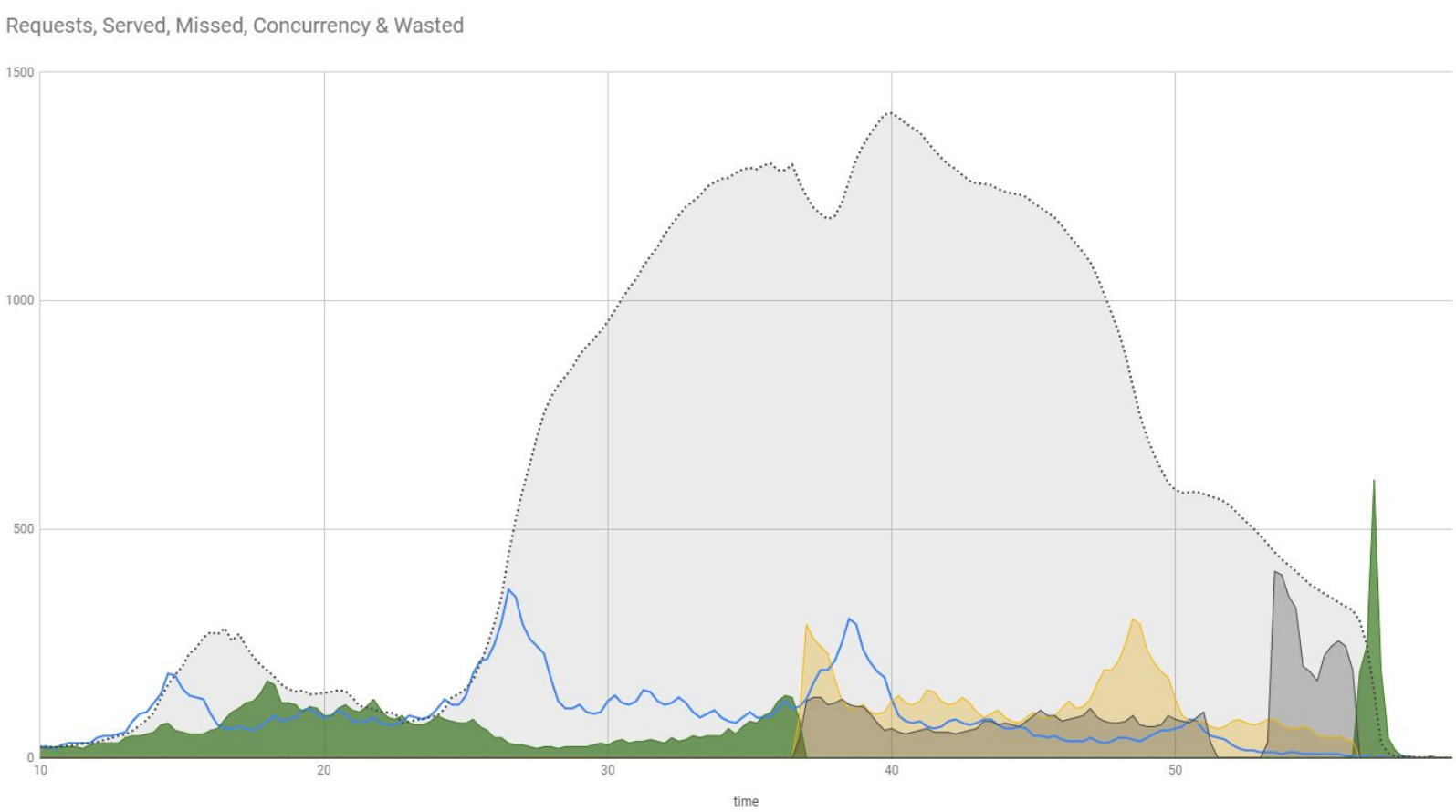


Controlled concurrency

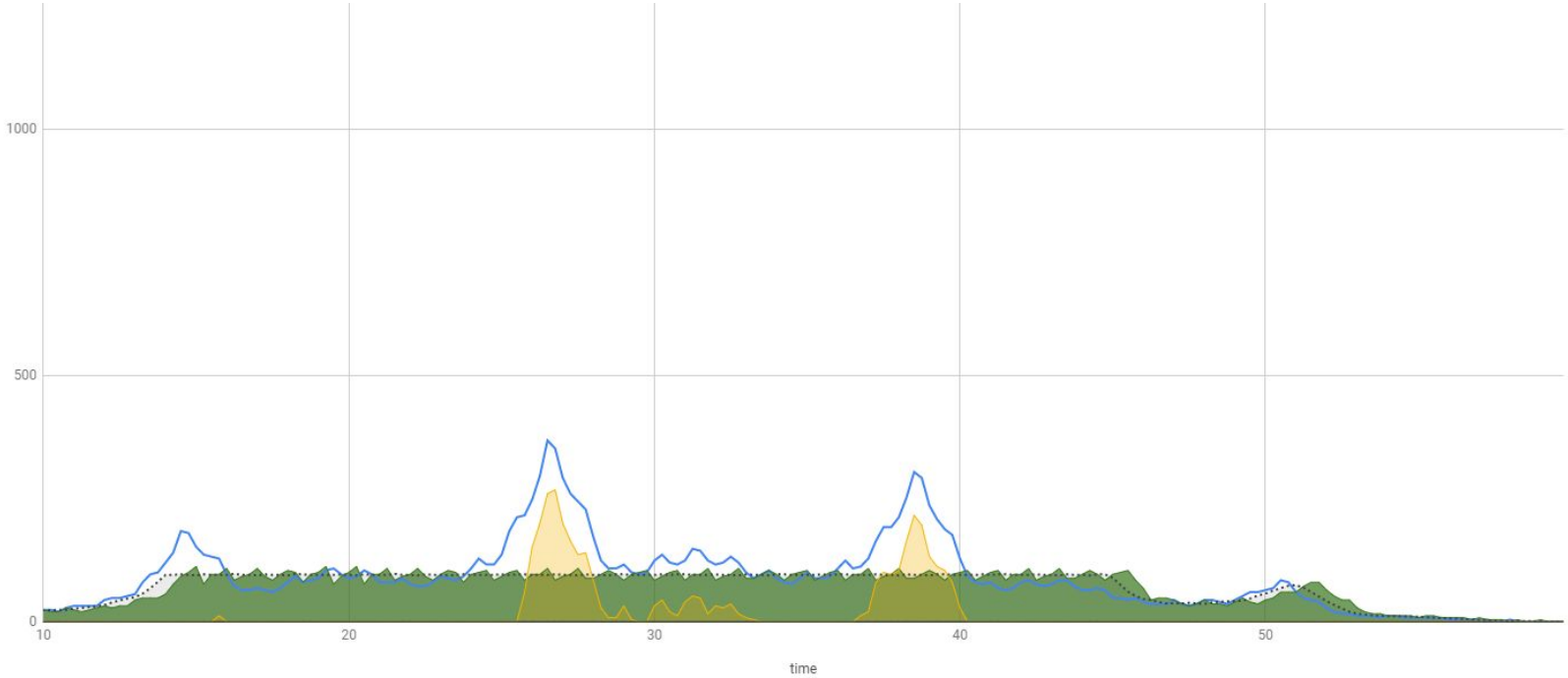


Bulkhead

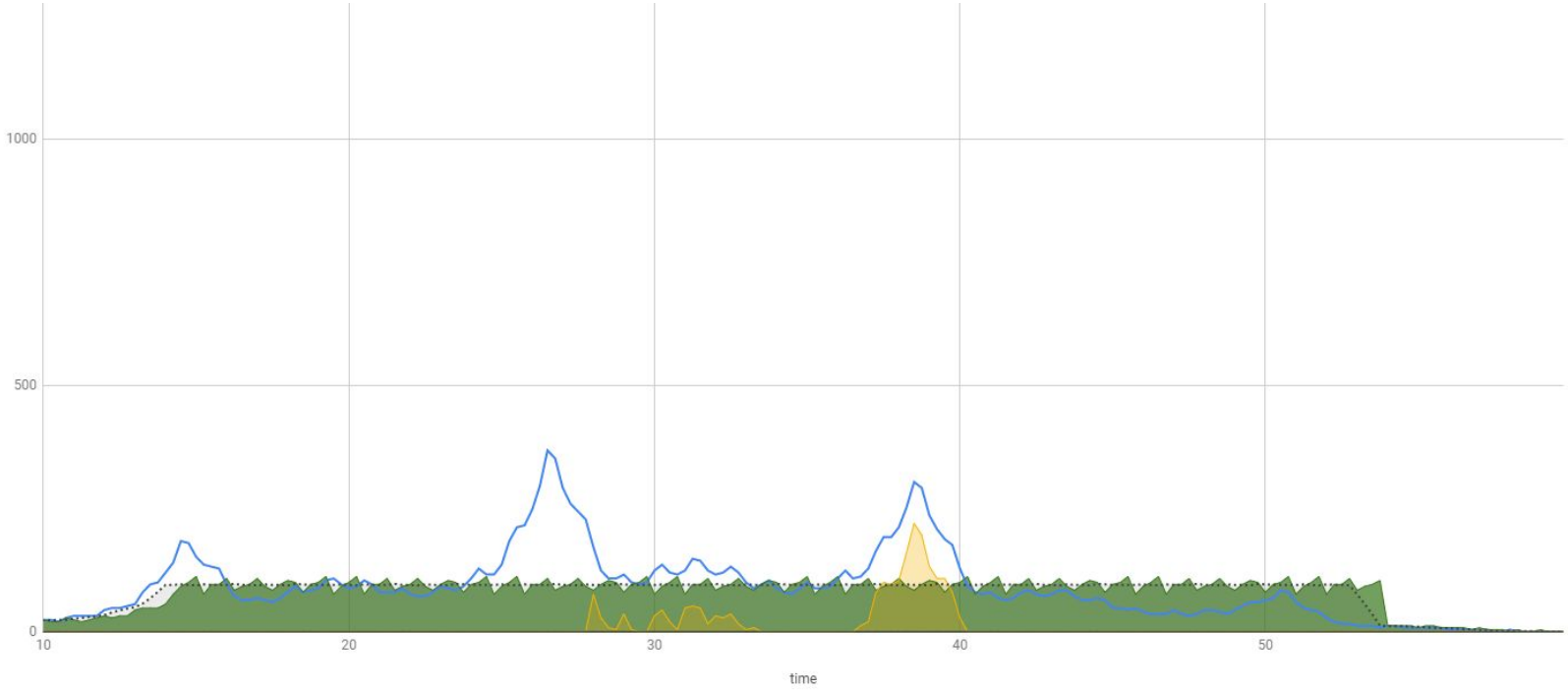
```
const int MaxConcurrency = 100;
SemaphoreSlim bulkhead = new SemaphoreSlim(MaxConcurrency, MaxConcurrency);

public async Task ProcessRequest()
{
    if (!await bulkhead.WaitAsync(TimeSpan.FromSeconds(1.0)))
    {
        throw new OperationCanceledException();
    }
    try { await ProcessRequestInternal(); return; }
    finally { bulkhead.Release(); }
}
```


Requests, Served, Missed, Concurrency & Wasted
1500
1000
500
0
10
20
30
40
50
time

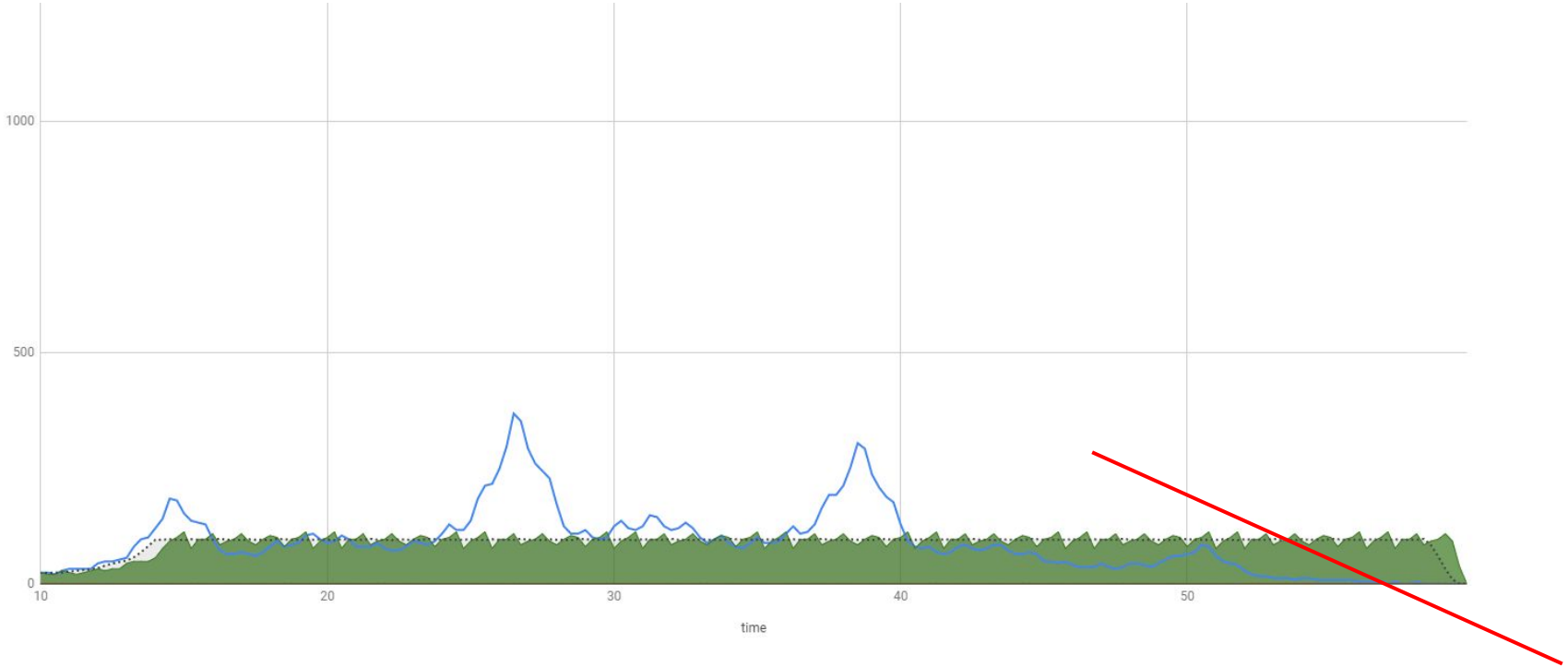
100 inside, 100 outside



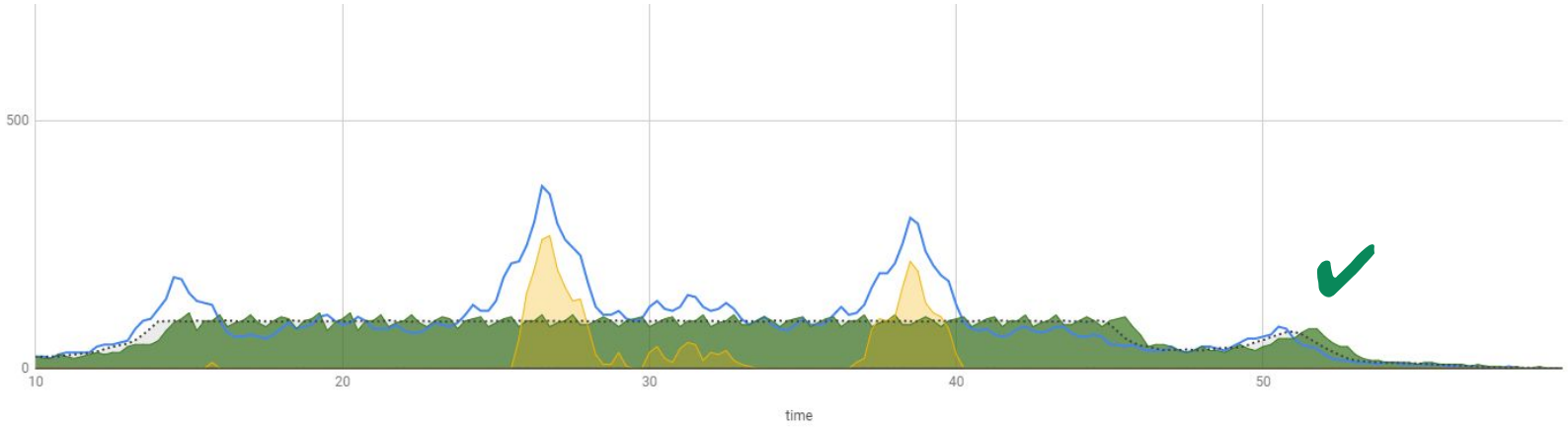
100 inside, 500 outside



100 inside, 1000 outside



100 inside, 100 outside

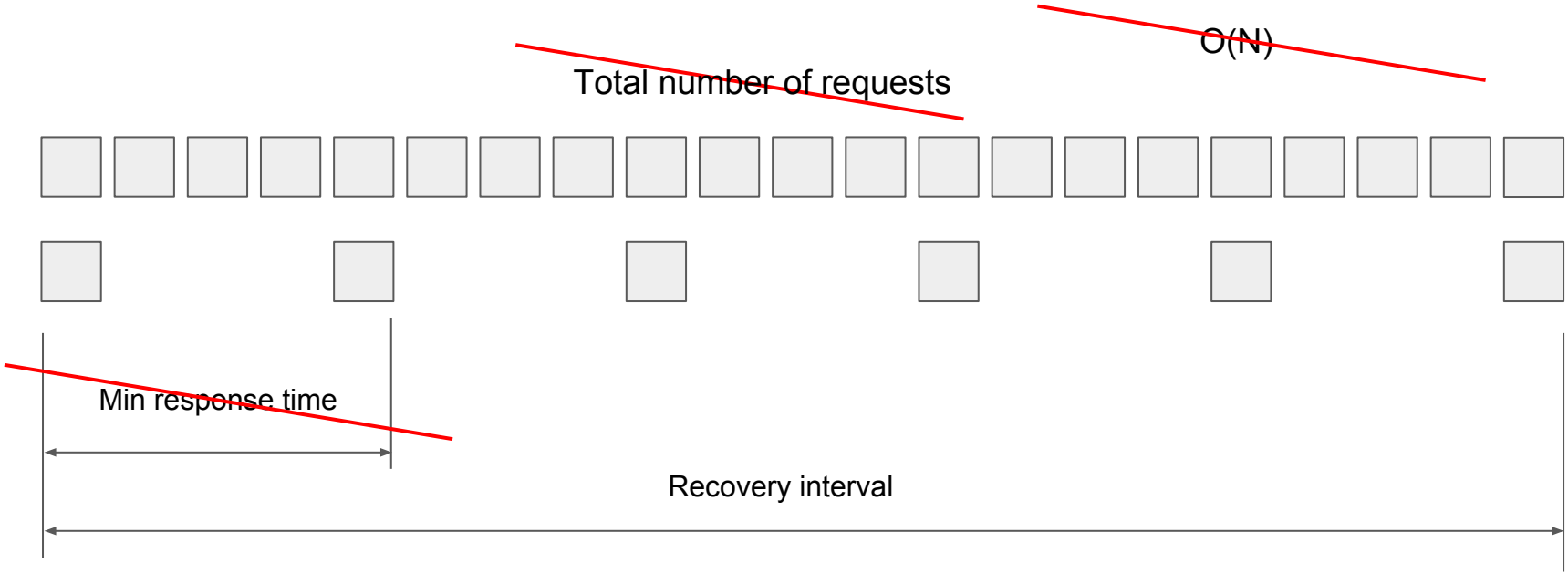


Asynchronous failures

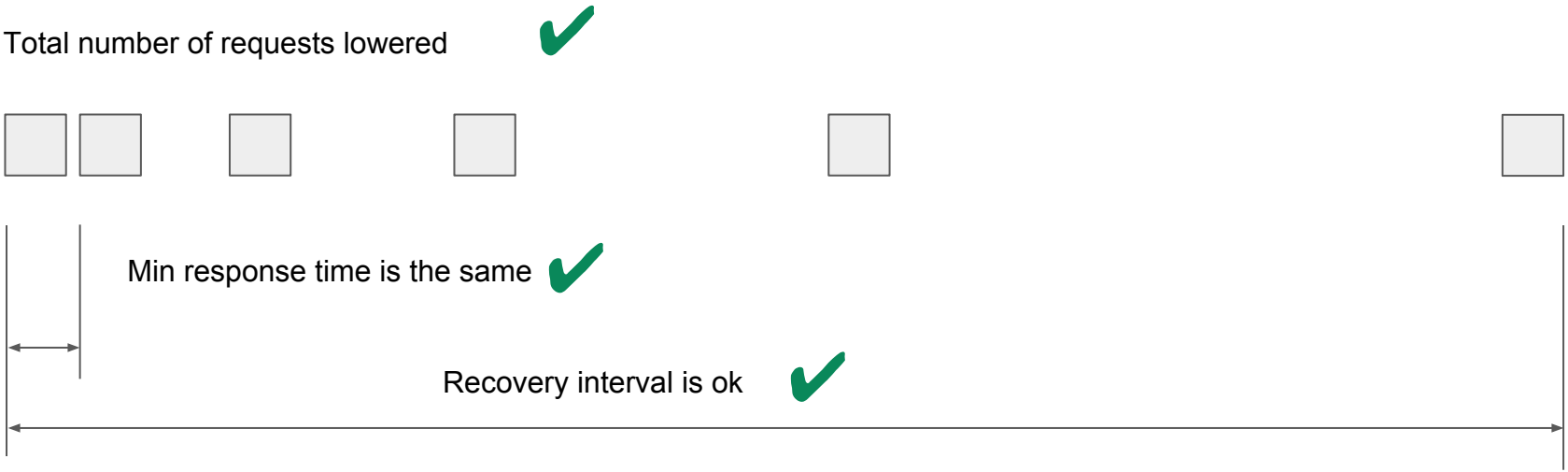
Cause: Uncontrolled Concurrency

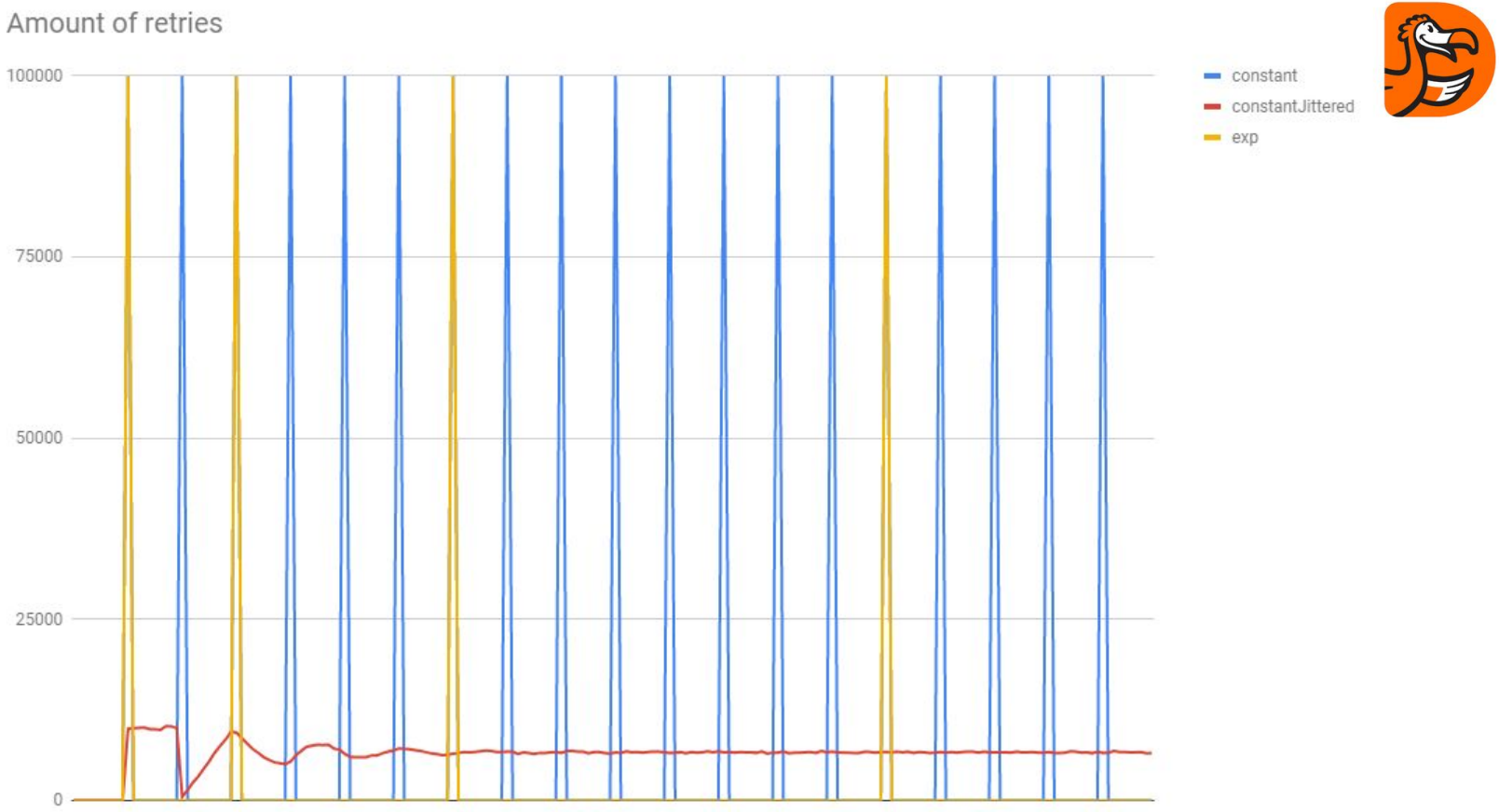
Solution: Bulkhead Isolation

Retry induced failure



Lowering the number of requests




Amount of retries
100000
75000
50000
25000
0
constant
constantJittered
exp

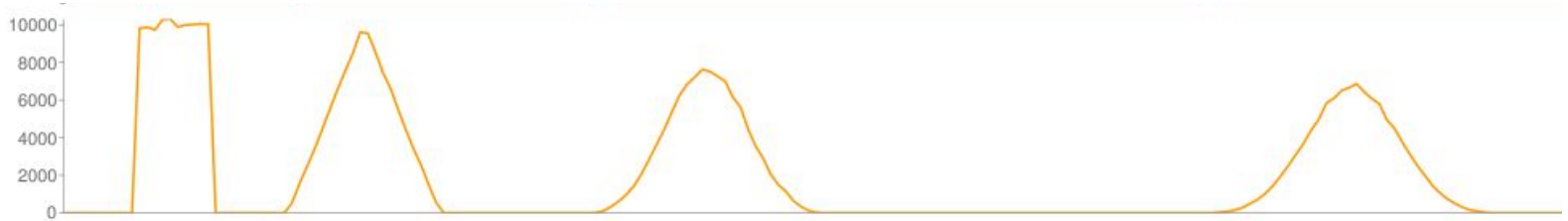
Exponential Backoff

```
Policy
    .Handle<HttpRequestException>()
    .WaitAndRetry(5,
        retryAttempt =>
            TimeSpan.FromSeconds(
                Math.Pow(2, retryAttempt))
                + TimeSpan.FromMilliseconds(random.Next(0, 1000));
```

Fixing concurrency spikes

```
i => Math.Pow(2, i)
```

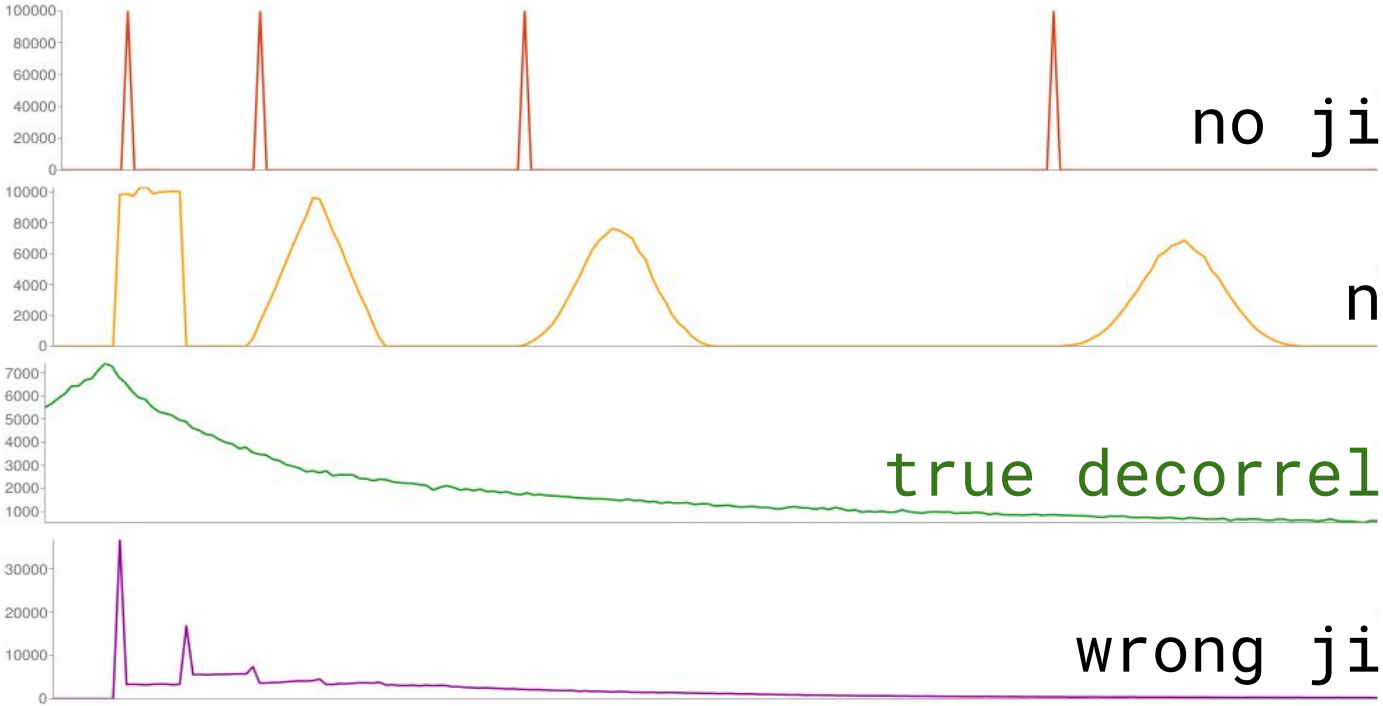

```
i => random.NextDouble() + Math.Pow(2, i)
```



Decorrelated exponential backoff ✔

```
IEnumerable<TimeSpan> DecorrelatedExponent()
{
    for (var softCount = 0.0; softCount < n; ) {
        softCount += random.NextDouble() * 2;
        yield return TimeSpan.FromSeconds(
            Math.Pow(2, softCount) * random.NextDouble());
```

Randomization is not trivial



no jitter 100

naive 10

true decorrelated 7

wrong jitter 30

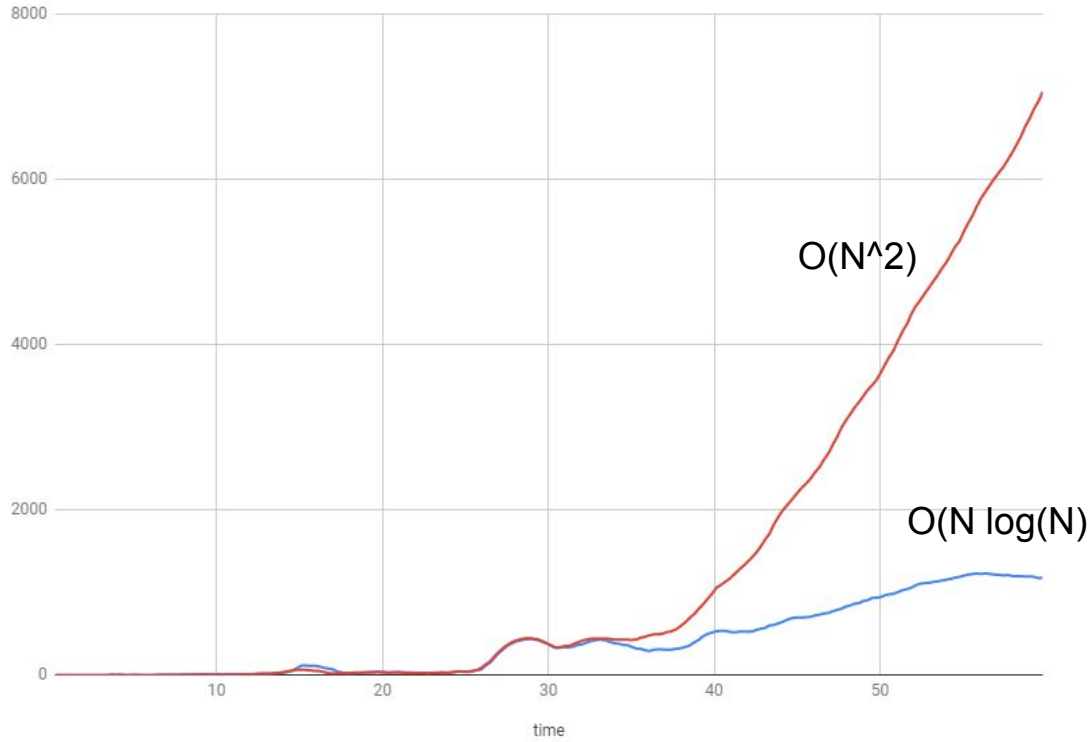
Calculating total time of exponentail retry

$$\sum_{k=1}^{n} ar^{k-1} = \frac{a(1-r^n)}{1-r}.$$

total = first (2^n - 1)

first = total / (2^n - 1)

Concurrency is lower using Exponential Backoff



Retry induced failure

Cause:

- Going beyond SLA by the client

Solution:

- Exponential Retry Policy
- Circuit Breaker

Complex policy using Polly

```
sc.AddHttpClient<IService, Client>(client => { client.Timeout =
settings.TimeOutPerRequest; })
.AddPolicyHandler(
    Policy
    .TimeoutAsync<HttpResponseMessage>(settings.TotalTimeOut))
.AddPolicyHandler(
    HttpPolicyExtensions
    .HandleTransientHttpError()
    .Or<TimeoutRejectedException>()
    .WaitAndRetryAsync(settings.RetryCount,
        i => TimeSpan
        .FromMilliseconds(20 * Math.Pow(2, i))))
// …
```

Complex policy using Polly

```
// …
.AddPolicyHandler(
    Policy
    .TimeoutAsync<HttpResponseMessage>(settings.TimeOutPerRequest))
.AddPolicyHandler(
    HttpPolicyExtensions
    .HandleTransientHttpError()
    .AdvancedCircuitBreakerAsync(
        settings.FailureThreshold,
        settings.SamplingDuration,
        settings.MinimumThroughput,
        settings.DurationOfBreak));
```

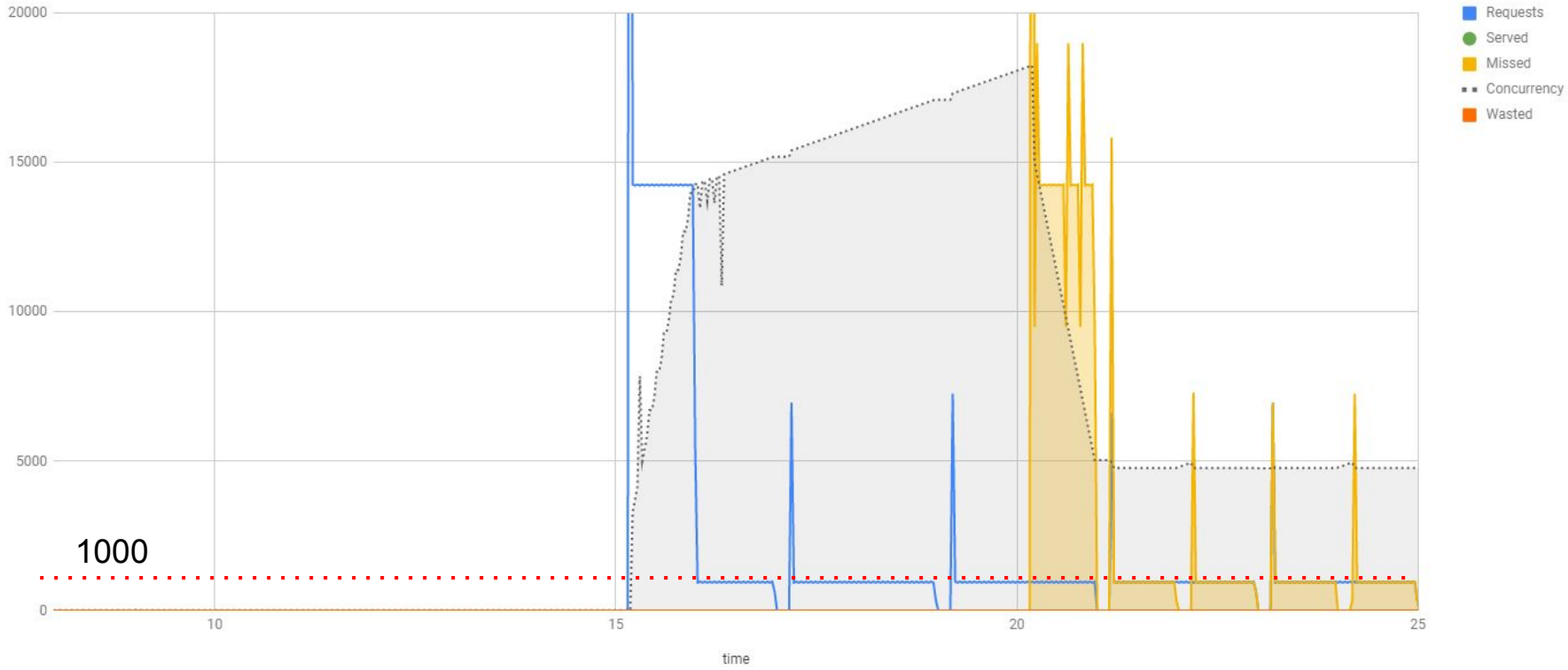
Cascading failure

Cause: Lack of Autonomity

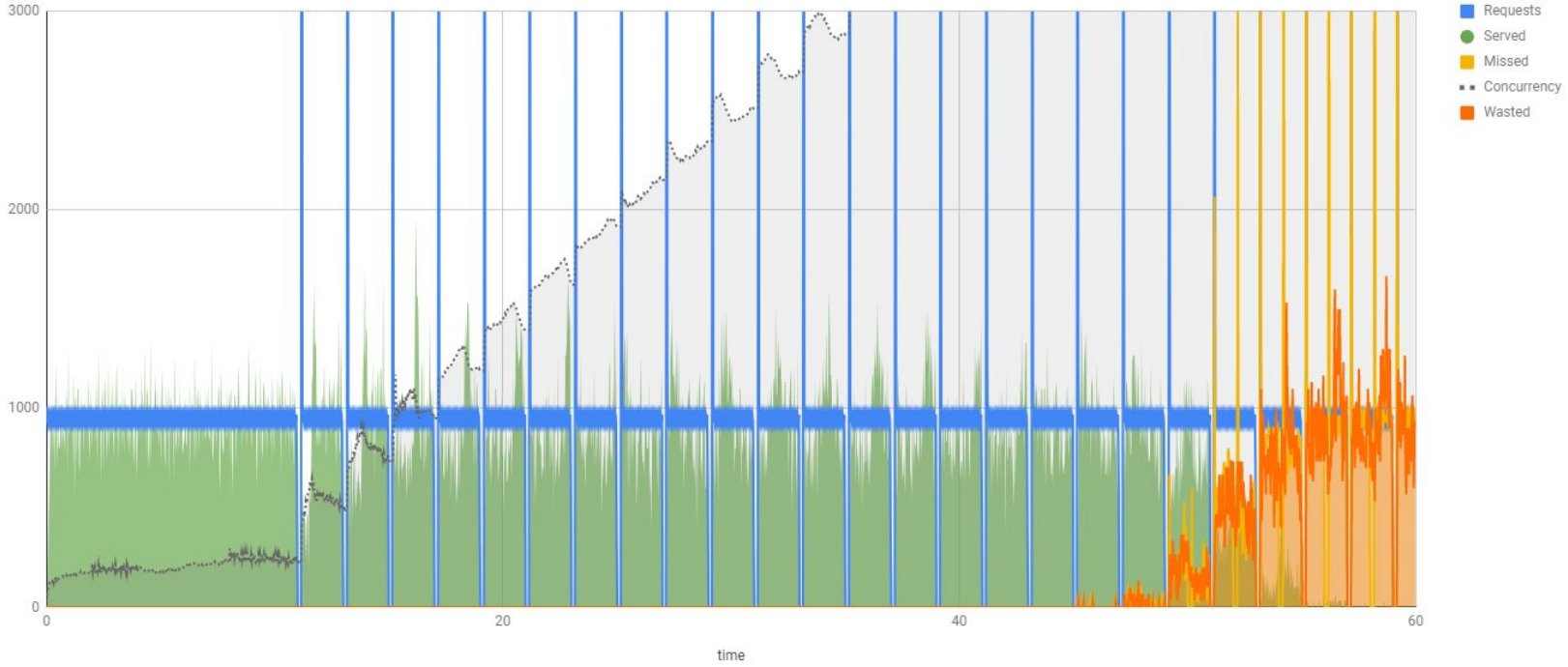
Solutions:

- True Autonomity,
- Fallback, Cache

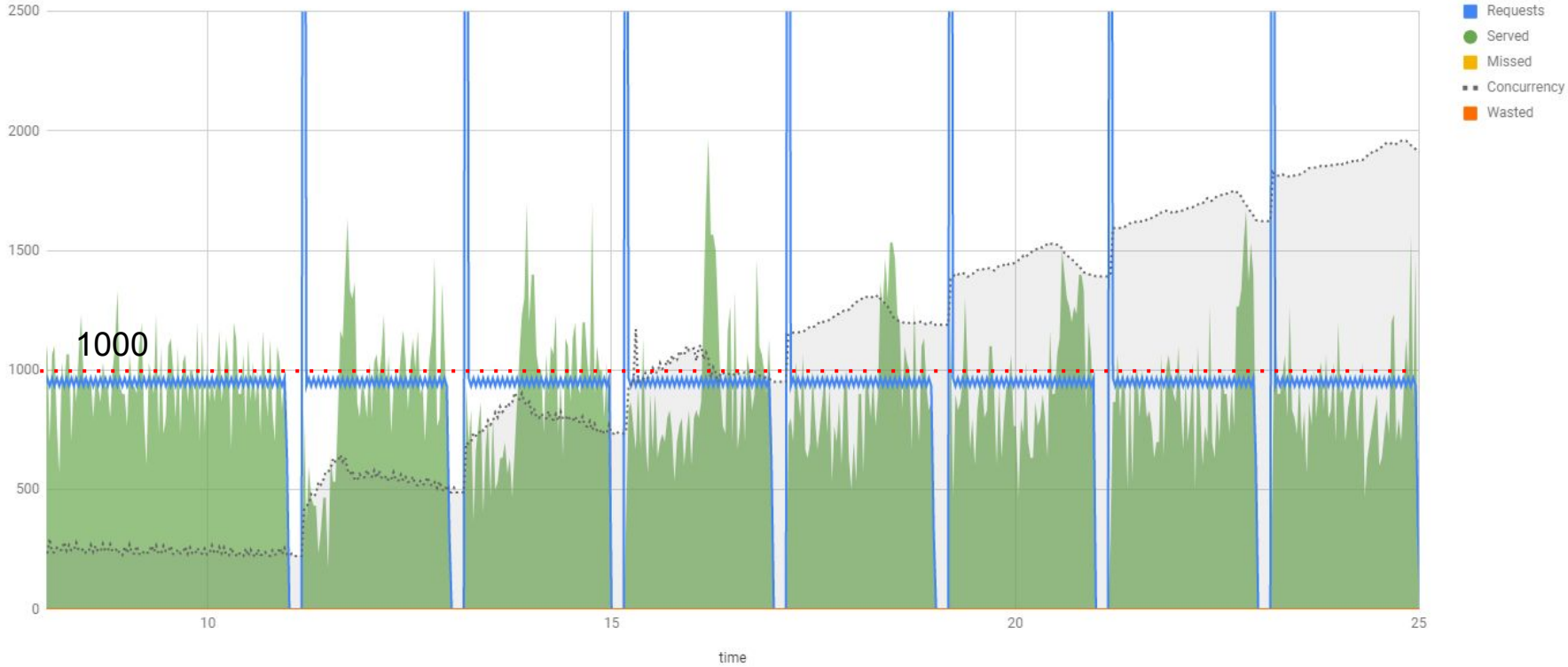
Cold Start failure



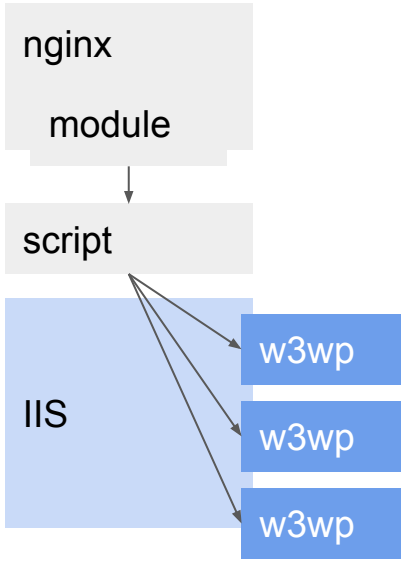
Beware - concurrency accumulates!

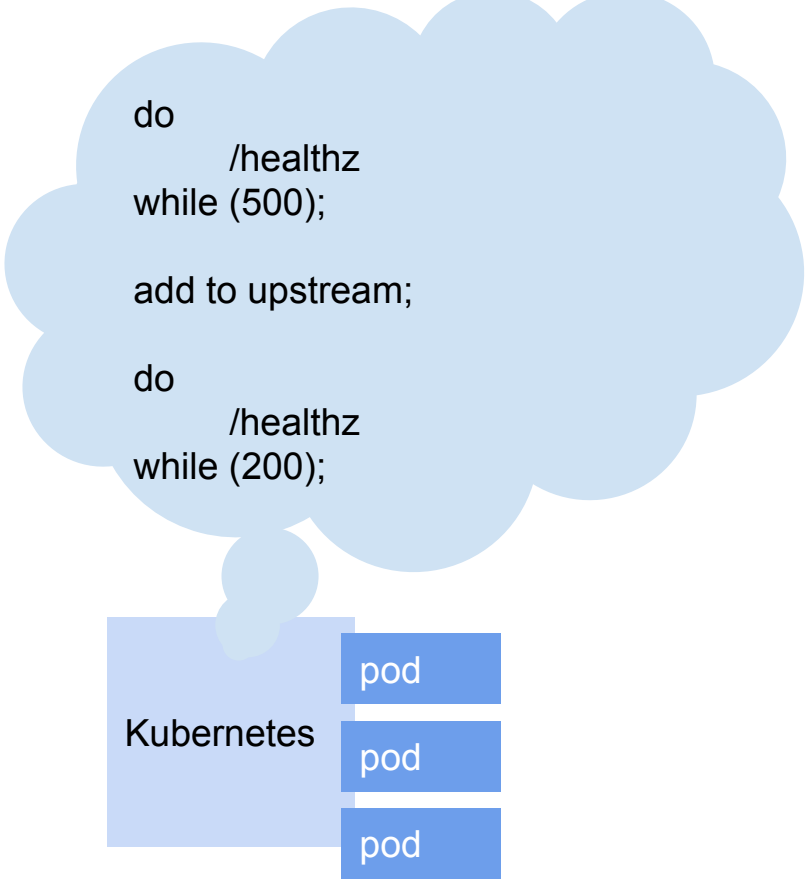


Beware - concurrency accumulates!



Readiness probe





```yaml
livenessProbe:
  httpGet:
    path: /healthz
    port: http
  initialDelaySeconds: 10
  timeoutSeconds: 5
  periodSeconds: 10
readinessProbe:
  httpGet:
    path: /healthz
    port: http
  initialDelaySeconds: 10
  timeoutSeconds: 5
  periodSeconds: 10
  failureThreshold: 6
```

```
[Route("healthz")]
0 references
public async Task<ActionResult> Healthz()
{
    var result = await CheckCached();
    return StatusCode(result.IsSuccessful() ? 200 : 500, result);
}

1 reference
async Task<TestResults> CheckCached()
{
```

Cold Start failure

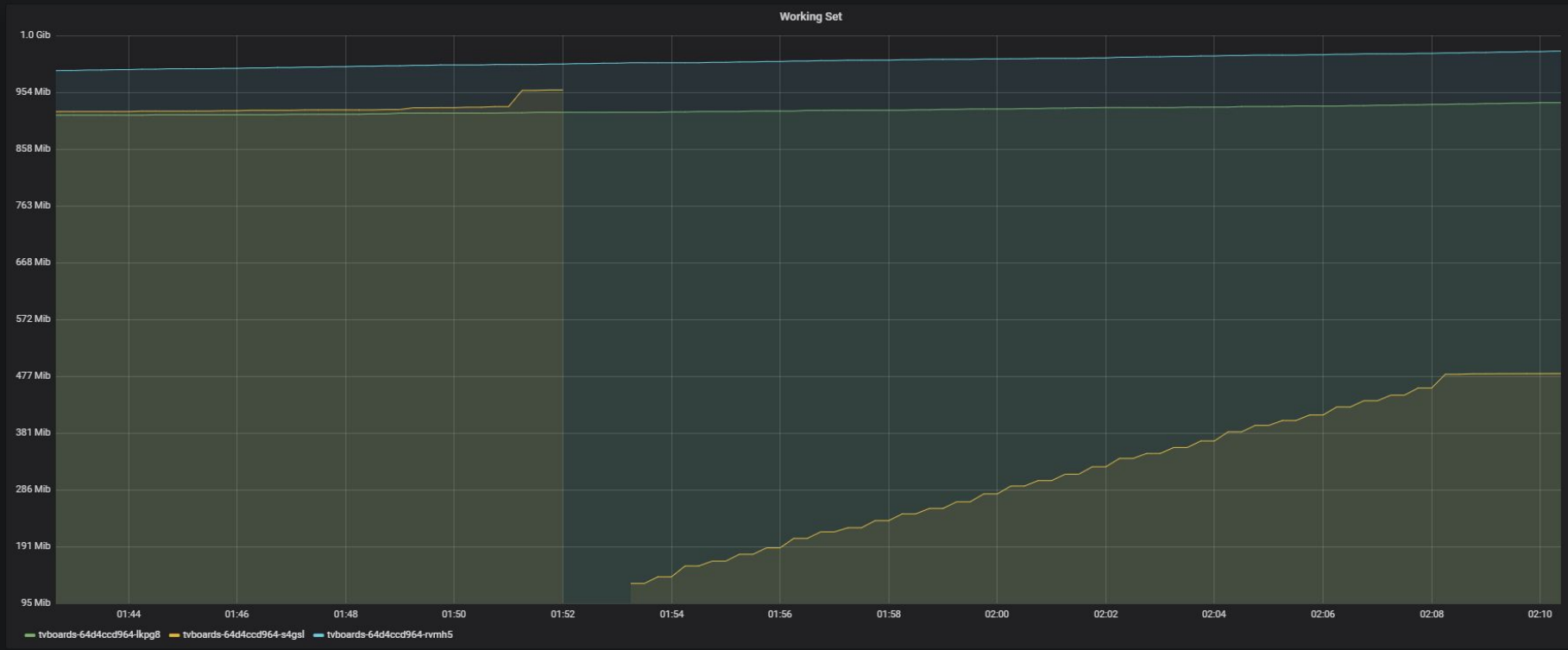
Cause: Lack of deployment strategy

Solution:

- IIS + nginx module
- k8s + Readiness probe
- Bulkhead

WE Cluster - TVBoards

Oct 28, 2018 01:42:39 to Oct 28, 2018 02:10:22




THIS IS FINE
Dodo Pizza Engineering © 2018
dodois.com

Usual suspects

Common

Concurrency, Contention, Algorithmic Complexity, SLA Violation, Fluctuations

Synchronous

Concurrency Model (Preemptive Multitasking)

Asynchronous

Uncontrolled Concurrency

Retry

SLA Violation

Cascade

Lack of Autonomity

Cold Start

Lack of Deployment Strategy

Performance of a distributed system

Fault Tolerance / Resilience

Subjected to:

- Concurrency
- Contention

Managing performance characteristics

Transient Fault Handling primitives (Polly)

Concurrency model

Requires:

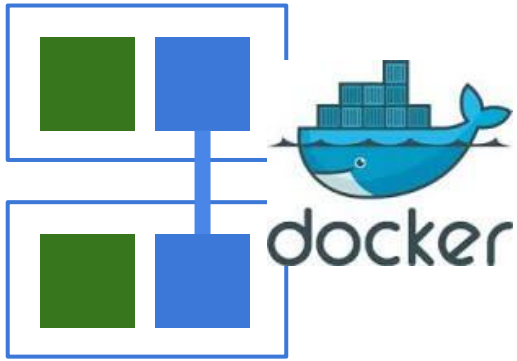
- Orchestration
- CI/CD
- Monitoring

Next steps

Global Policy / Middleware

```
app.Use(Middleware)
```

Kubernetes Service Mesh



Next steps

Messaging architectures

Serverless

Event sourcing

3-factor apps

References

Concurrency is not Parallelism. Rob Pike https://blog.golang.org/concurrency-is-not-parallelism

Polly .**NET** resilience and transient-fault-handling library https://github.com/App-vNext/Polly

Polly is now integrated with ASPNET https://docs.microsoft.com/en-us/aspnet/core/fundamentals/http-requests?view=aspnetcore-2.1

There Is No Thread. Stephen Cleary https://blog.stephencleary.com/2013/11/there-is-no-thread.html

Understanding Azure Virtual Machine IOPS, throughput and disk latency – Part 1

https://blogs.technet.microsoft.com/andrewc/2016/09/09/understanding-azure-virtual-machine-iops-throughput-and-disk-latency/

Exponential Backoff And Jitter https://aws.amazon.com/ru/blogs/architecture/exponential-backoff-and-jitter/

Performance Architecture of Dodo IS

Engineering team at Dodo Pizza welcomes your feedback!

George Polevoy

Reliability Engineer, Dodo Pizza
g.polevoi@dodopizza.com

https://www.facebook.com/dodopizzaio/