# 背景

在网络系统中，限流器被用于控制客户端或服务端发送流量的速率。在HTTP世界中，限流器限制在指定时间内允许发送客户端请求数。如果API请求次数超过了限流器设置的阈值，则所有超出的调用都会被阻止。以下是一些示例：

单个用户每秒最多可以发布2个帖子

每天同一IP地址最多可以创建10个帐户。

同一设备每周最多可以领取5次奖励。

在本章中，您需要设计一个限流器。在开始设计之前，我们先看看使用 API 限流器的好处：

防止拒绝服务 (DoS) 攻击造成的资源匮乏[1]。几乎大型科技公司发布的所有 API 都强制执行某种形式的速率限制。例如，Twitter将每3小时的推文数量限制为300条[2]。Google docs APIs有如下默认限制：每个用户每60秒读取请求为300次[3]。限流器通过阻止多余的调用来防止DoS攻击，无论是有意还是无意的。

降低成本。限制过多的请求意味着减少服务器数量，将更多资源分配给高优先级的API。速率限制对于使用付费第三方API的公司来说极为重要。例如，你对以下外部API的调用是按次数收费的：检查信用、付款、检索健康记录等。限制调用次数对减少成本至关重要。

防止服务器过载。为了减少服务器的负荷，使用限流器来过滤由机器人或用户的不当行为造成的过量请求。

# 方案

## 方案设计

**要求：**

准确地限制过多的请求

低延迟：限流器不能降低Http响应时间。

尽可能的占用更少的内存。

分布式速率限制，要求可以在多个服务器或进程之间共享。

异常处理，当用户的请求受到限制时，向用户显示明确的异常信息。

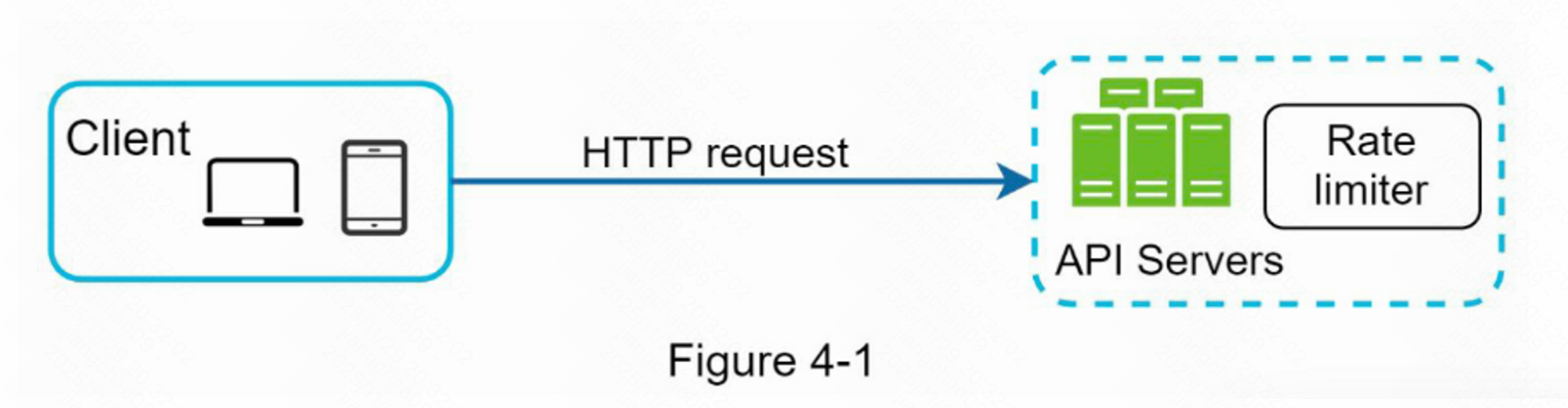
高容错性。如果限流器出现任何问题（例如，一个缓存服务器离线），它不会影响整个系统。

**限流器放在哪里？**

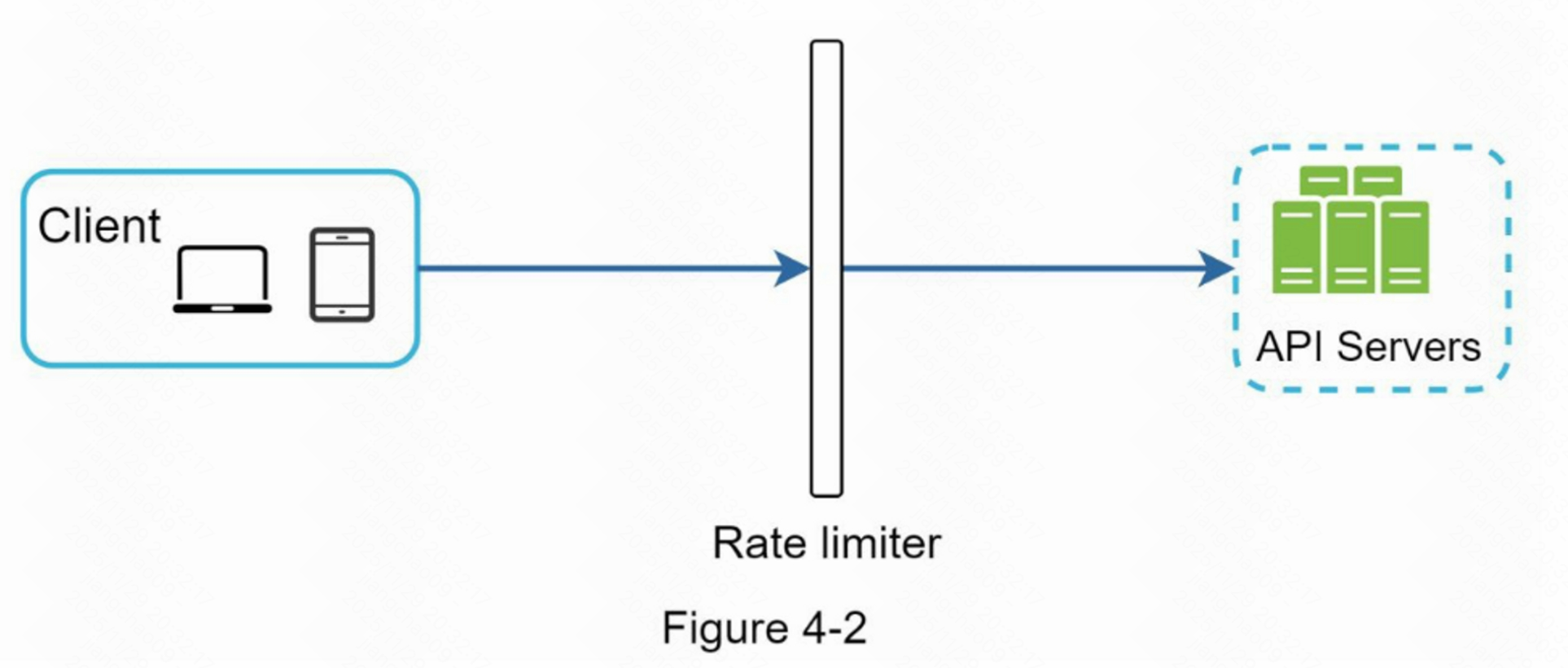
你可以在客户端或服务器端实现限流器。

客户端实现。一般来说，客户端是执行速率限制的不可靠场所，因为客户端请求很容易被恶意用户伪造。此外，我们可能无法控制客户端的实现。

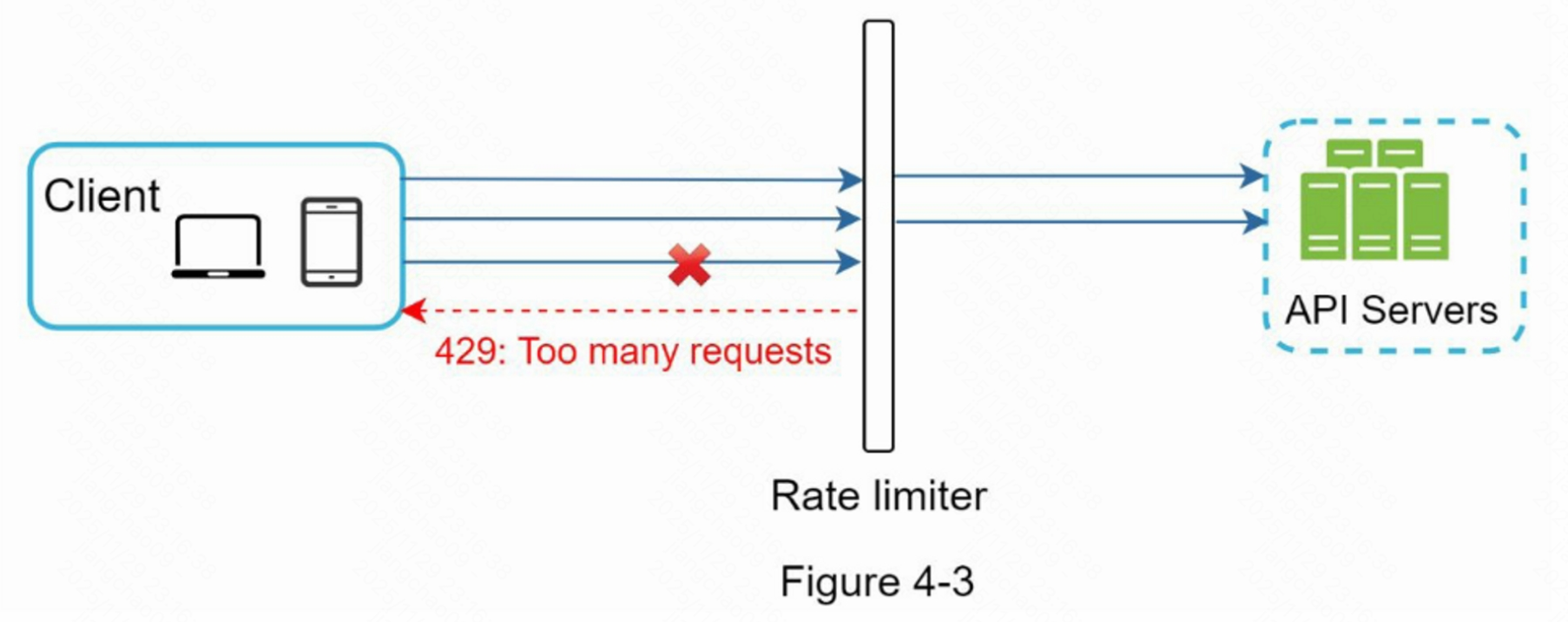
服务端实现。图4-1显示了一个放在服务器端的限流器。



除了客户端和服务器端的实现方式之外，还有另一种方法。我们可以不将限流器放在API服务器上，而是创建一个限流器中间件，用于对API的请求进行限制，如图 4-2 所示：



让我们使用图 4-3 中的示例来说明这种设计中的限流器是如何工作的。假设我们的 API 允许每秒 2 个请求，而客户端在一秒内发送了3个请求到服务器。前两个请求被路由到 API 服务器。但是，限流器中间件会限制第三个，请求并返回 HTTP 状态代码 429。HTTP 429 响应状态代码表示用户发送了过多的请求。



云微服务[4]已经变得非常流行，限流通常在一个叫做API网关的组件中实现。API网关是一个完全托管的服务，支持限速、SSL、认证、IP白名单、服务静态内容等。现在，我们只需要知道API网关是一个支持限流的中间件。

在设计限流器时，要问自己的一个重要问题是：限流器应该在哪里实现，在服务器端还是在网关中？这没有绝对的答案。这取决于你公司目前的技术栈、工程资源、优先级、目标等。这里有一些通用的指南：

评估你目前的技术栈，如编程语言、缓存服务等。确保你目前的编程语言能够有效地在服务器端实现速率限制。

确定适合你业务需求的速率限制算法。当你在服务器端实施一切时，你可以完全控制算法。然而，如果你使用第三方网关，你的选择可能是有限的。

如果你已经使用了微服务架构，并在设计中包含了一个API网关来执行认证、IP白名单等，你可以在API网关上添加一个限流器。

建立你自己的速率限制服务需要时间。如果你没有足够的工程资源来实现限流器，商业API网关是一个更好的选择。

## 限流算法

限流可以使用不同的算法来实现，每种算法都有其独特的优缺点。尽管本章并不关注算法，但在高层次上了解它们有助于选择正确的算法或算法组合来适应我们的使用情况。

下面是一个流行算法的列表：

令牌桶（Token bucket）

漏桶算法（Leaking bucket）

固定窗口计数器（Fixed window counter）

滑动窗口日志（Sliding window log）

滑动窗口计数器（Sliding window counter）

### 漏桶算法

漏桶（Leaking bucket）算法与令牌桶类似，不同之处在于请求是以固定速率处理的。它通常用先入先出（FIFO）队列来实现。

#### 原理

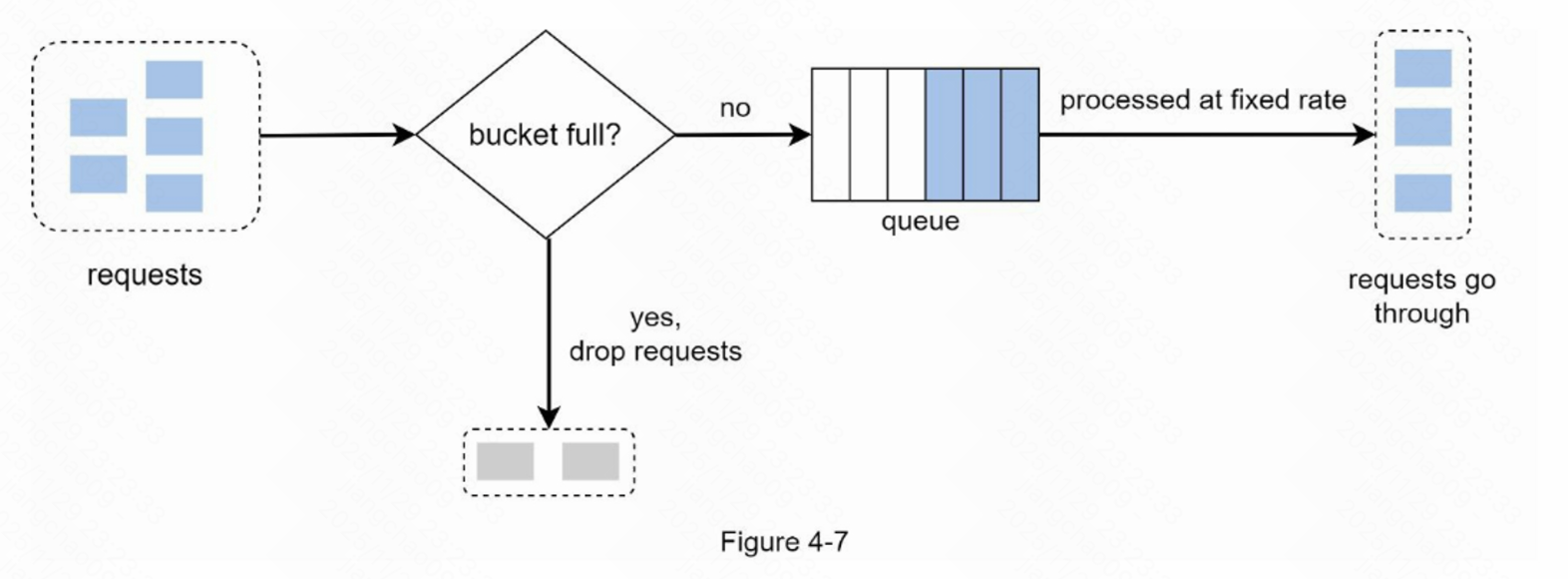
该算法的工作原理如下：

当请求到达时，系统会检查队列是否已满。如果队列未满，则将请求添加到队列中。

否则，将丢弃该请求。

请求会在固定的时间间隔内从队列中取出并进行处理。

图4-7解释了该算法的工作原理：



漏桶算法需要以下两个参数：

1、桶的大小：它等于队列的大小。队列容纳了要以固定速度处理的请求。

2、流出率：定义了在固定速率下可以处理多少请求，通常以秒为单位。

Shopify，一家电子商务公司，使用泄漏桶来限制速度[7]。

#### 优点

输出速率稳定可控：请求以固定的速率处理，因此它适用于需要稳定流出速率的用例。

避免流量抖动突增

对下游保护强

鉴于队列大小有限，内存效率高

#### 缺点

不支持突发（与令牌桶相反）

等待排队机制可能带来延迟：突发的流量使队列中充满了旧的请求，如果这些请求没有得到及时处理，最近的请求将受到速率限制。

算法中有两个参数，要适当地调整它们可能并不容易。

#### 适用场景

下游系统承受能力有限（数据库/第三方 API）

需要平滑流量（如视频直播平台、CDN）

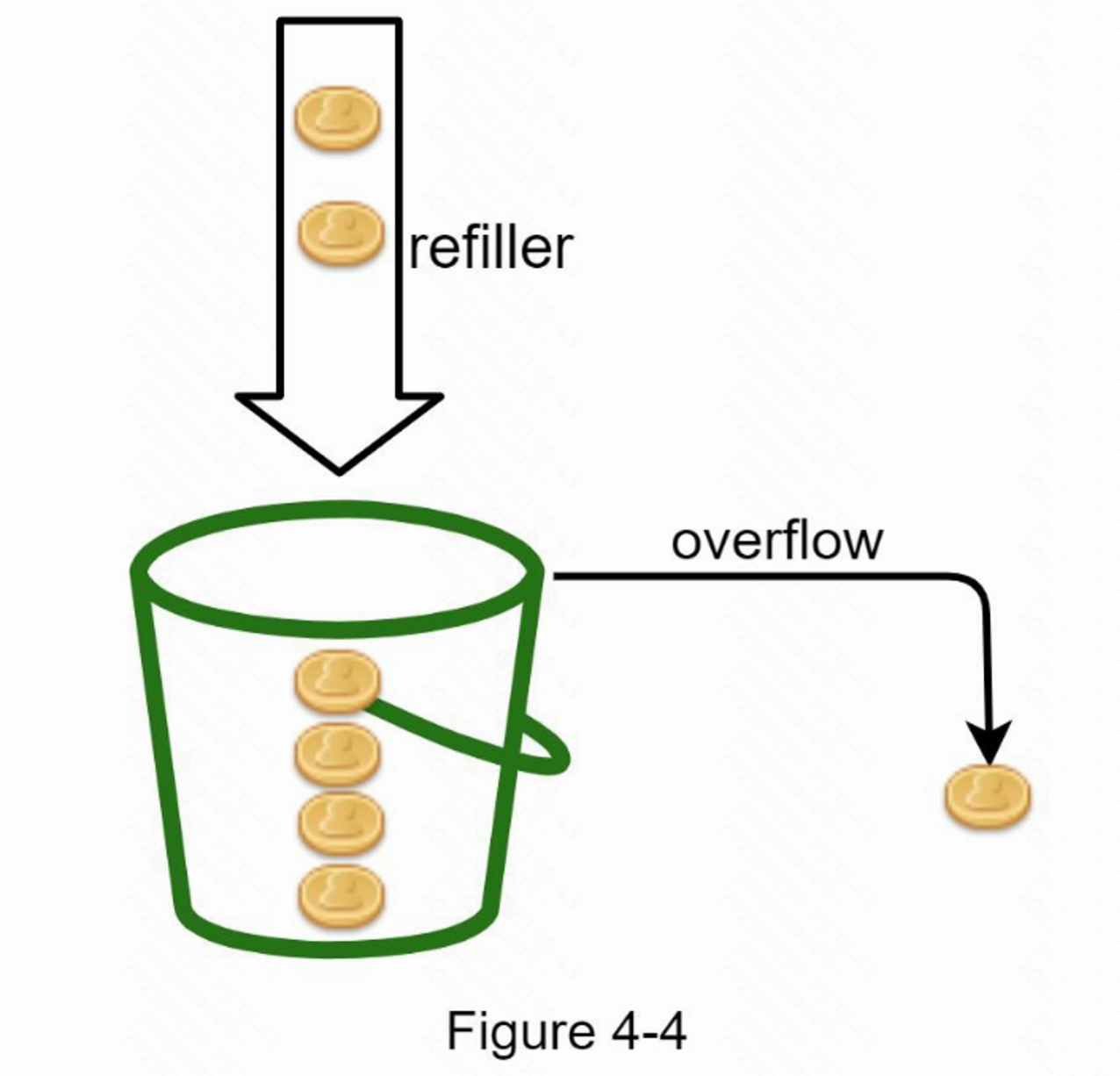
### 令牌桶算法

令牌桶（Token bucket）算法广泛用于速率限制。它简单易懂，常被互联网公司采用。Amazon [5]和Stripe [6]都使用这个算法来限制他们的 API 请求。

#### 原理

令牌桶算法的工作原理如下：

- 令牌桶是一个具有预定义容量的容器。令牌会以预定的速率定期放入桶中，一旦桶满了，就不再添加令牌。如图4-4所示，令牌桶容量为4，注入装置每秒向桶中放入2个令牌，一旦桶满了，多余的令牌就会溢出。



- 每个请求都会消耗一个令牌。当一个请求到达时，我们检查桶中是否有足够的令牌。图4-5解释了它是如何工作的。

如果有足够的令牌，我们会为每个请求取出一个令牌，然后请求通过。

如果没有足够的令牌，则该请求被丢弃。

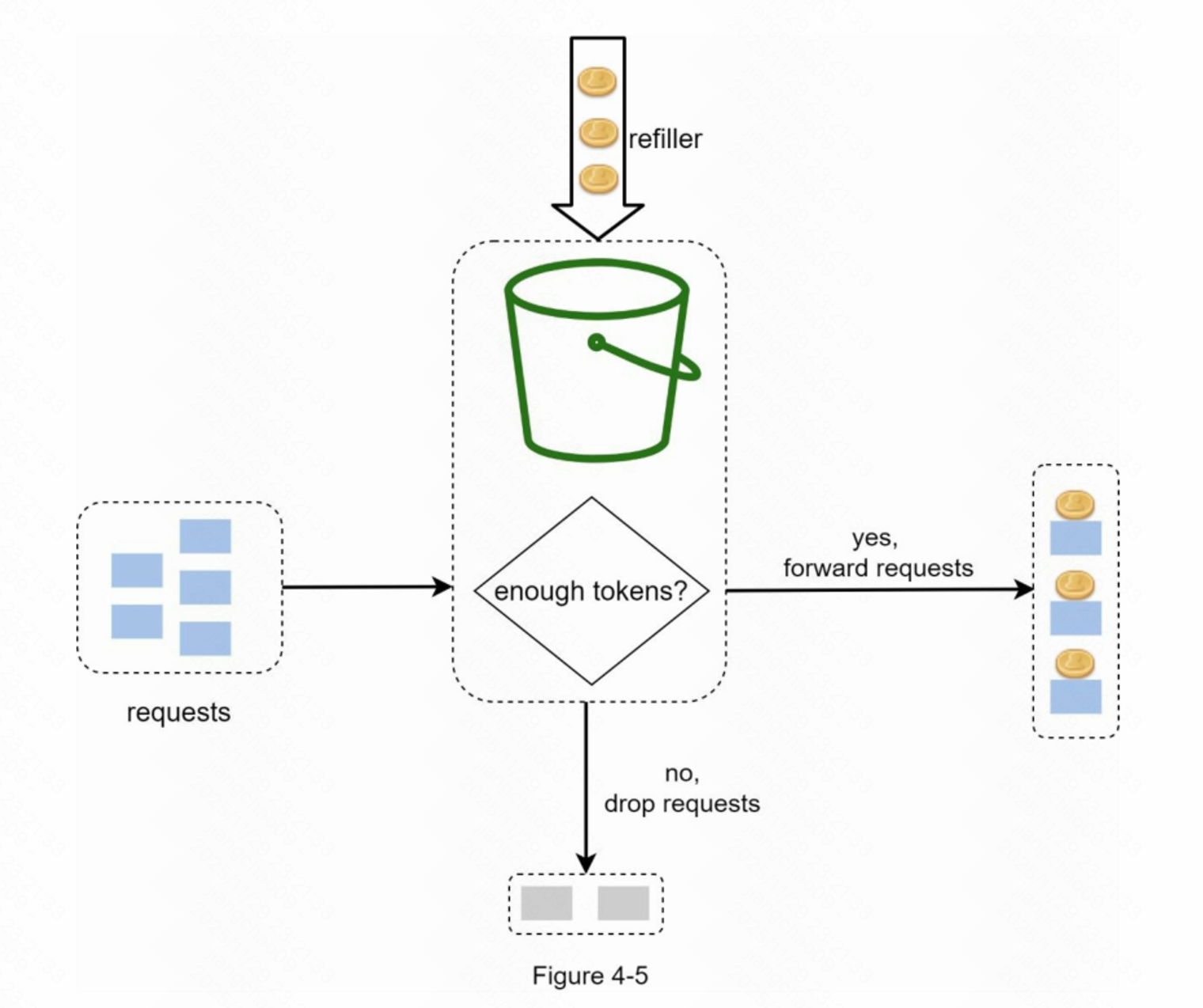
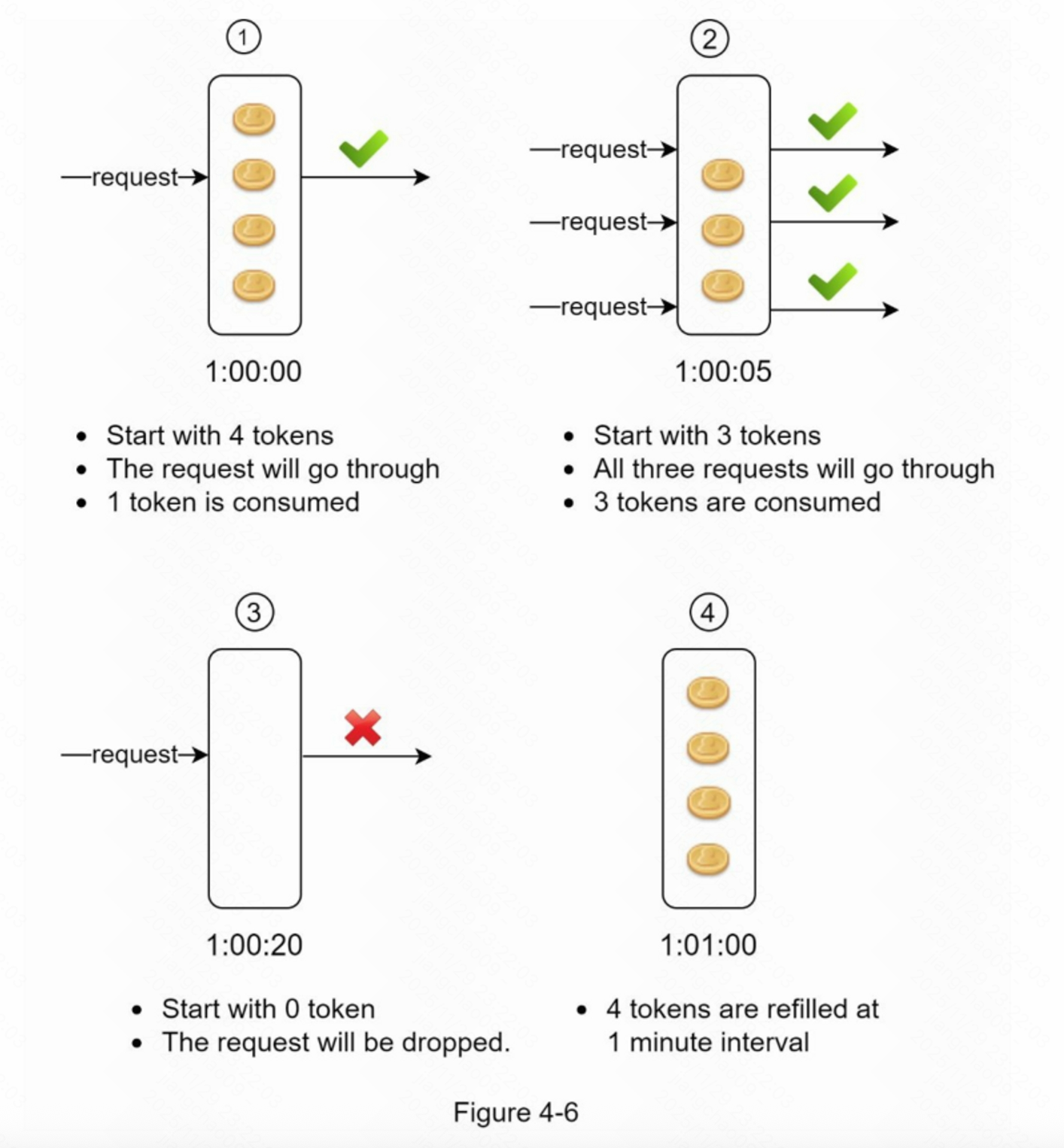


图4-6说明了令牌消耗、重新填充和速率限制逻辑的工作原理。在此示例中，令牌桶大小为4，重新填充速率为每1分钟4个。



令牌桶算法需要两个参数：

1、桶大小：桶内允许的最大令牌数。

2、填充速率：每秒放入到桶内的令牌数量。

我们需要多少个桶？这取决于限流规则，并且会有所不同。以下是几个例子。

通常需要为不同的API端点使用不同的桶。例如，如果一个用户被允许每秒发1个帖子，每天添加150个好友，并且每秒钟点赞5个帖子，则每个用户需要3个桶。

如果我们需要根据IP地址对请求进行限流，每个IP地址都需要一个桶。

如果系统允许每秒最多10,000个请求，则有一个全局桶供所有请求共享是有意义的。

#### 优点

算法容易实现

占用内存少

令牌桶允许在短时间内进行突发流量。只要有剩余的令牌，请求就可以通过。

#### 缺点

算法中有两个参数，即桶的大小和令牌的补充速率。然而，正确调整它们可能具有挑战性。

#### 适用场景

API Gateway/微服务入口限流

允许业务小峰值（如秒杀、推送）

实时系统需要平滑响应

### 固定窗口计数器算法

#### 原理

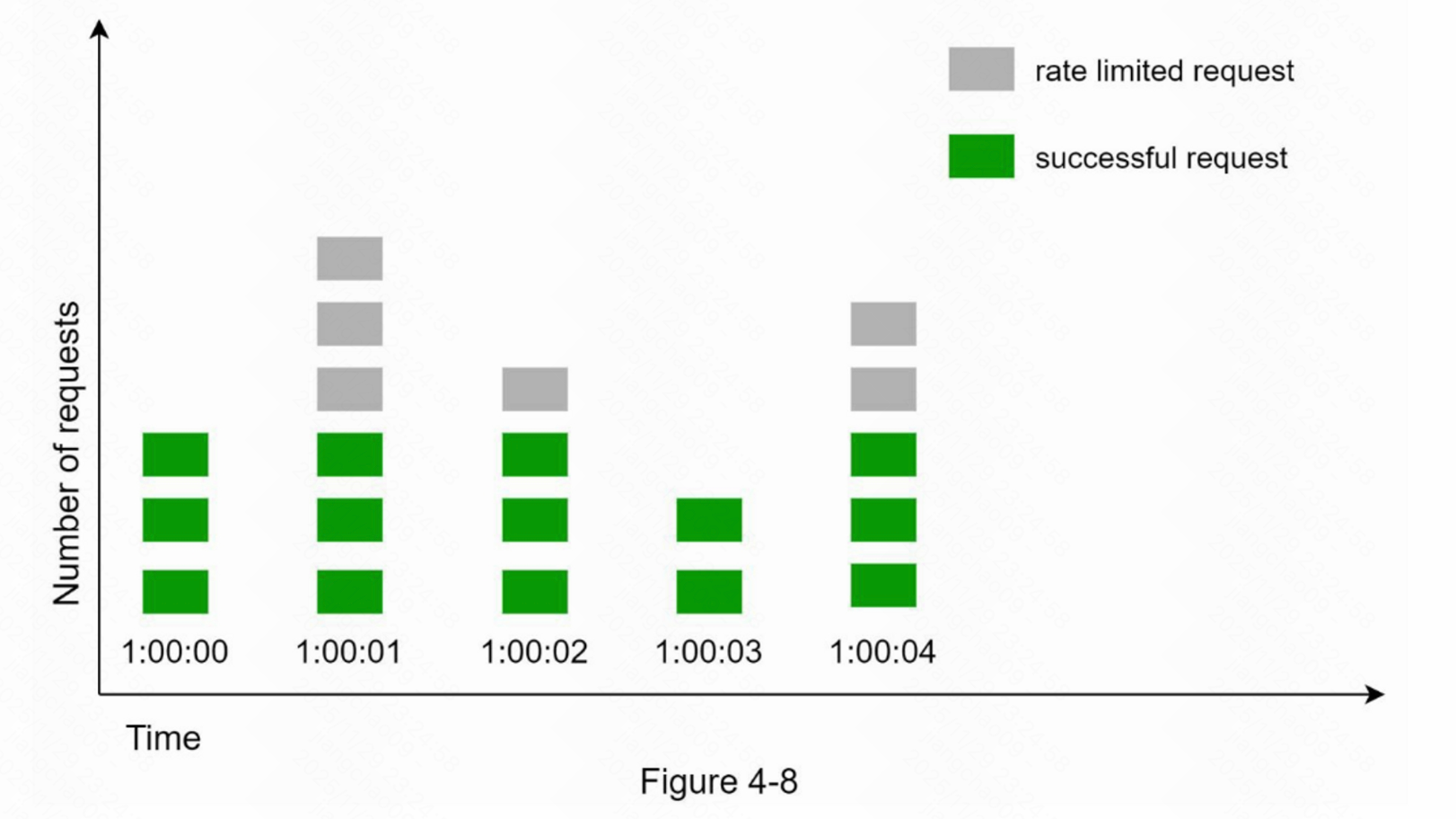
固定窗口计数器（Fixed window counter）算法工作原理如下：

该算法将时间轴划分为固定大小的时间窗口，并为每个窗口分配一个计数器。

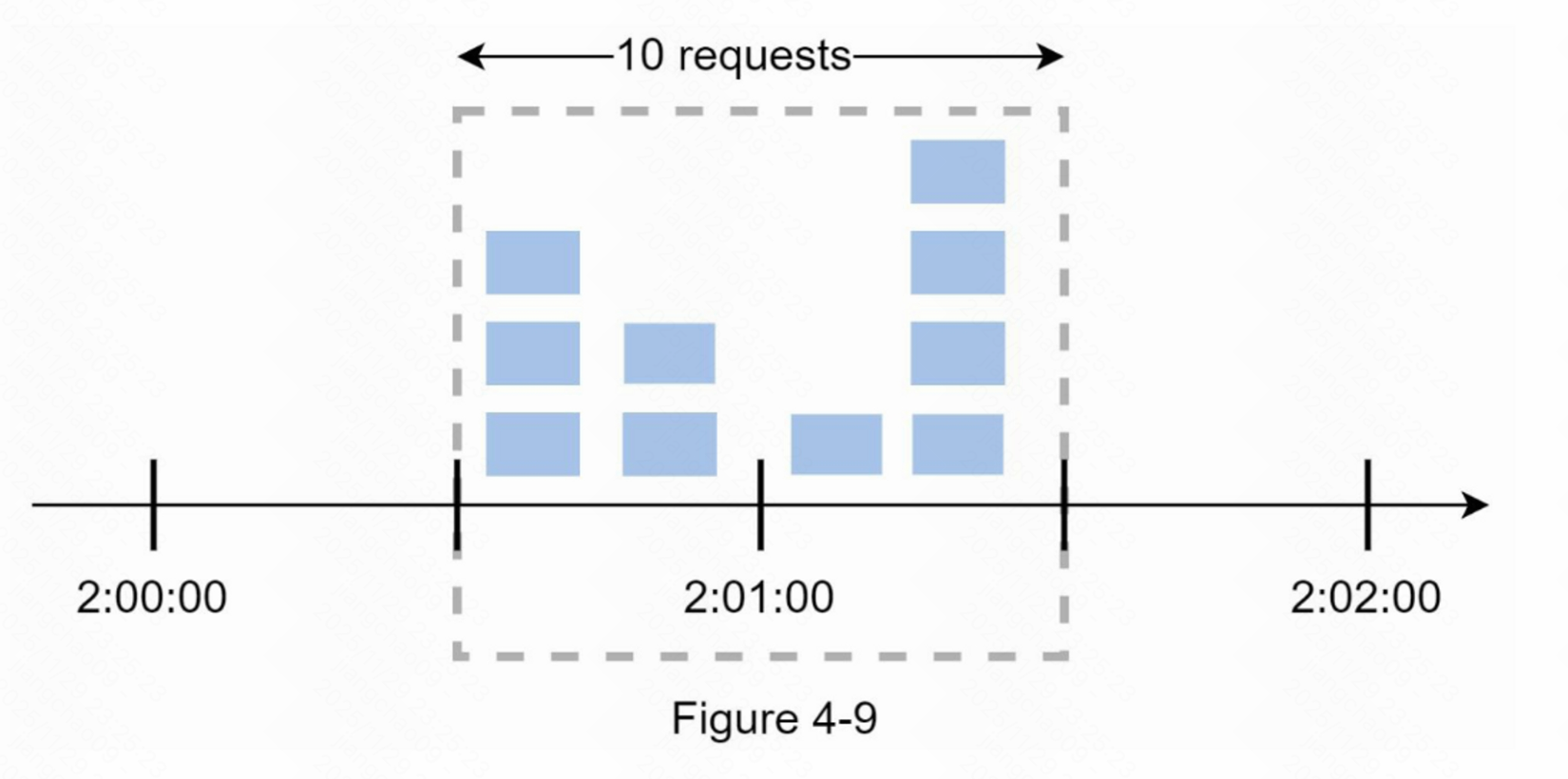
每个请求将计数器增加一。

一旦计数器达到预定的阈值，新的请求就会被放弃，直到新的时间窗口开始。

让我们用一个具体的例子来看看它是如何工作的。在图4-8中，时间单位是1秒，系统允许每秒钟最多有3个请求。在每个秒窗口中，如果收到的请求超过3个，额外的请求就会被放弃，如图4-8所示：



该算法的一个主要问题是，在时间窗口边缘的突发流量可能会导致超过允许配额的请求。考虑以下情况：



在图4-9中，系统允许每分钟最多有5个请求，可用的配额在整点分钟时重置。如图所示，在 2:00:00 和 2:01:00 之间有5个请求，在2:01:00和2:02:00 之间又有五个请求。在 2:00:30 和 2:01:30 之间的1分钟窗口，有10个请求通过了。这是允许请求数量的两倍。

#### 优点

实现简单，使用计数器即可

性能好，开销最低

内存高效

在单位时间窗口结束时重新设置可用配额，适合某些使用情况

#### 缺点

有“临界问题”（Boundary Problem），在窗口边界瞬间可能发生双倍请求，窗口边缘的流量激增可能导致超过允许配额的请求被通过（有突刺）。

如：

window1 最末0.01秒 100个请求 + window2 最初0.01秒 100个请求 = 200个

#### 适用场景

对突发高峰不敏感的后台接口

非严格限流的场合（例如后台管理接口）

### 滑动窗口日志算法（Sliding Log）

#### 原理

如前所述，固定窗口计数（Fixed window log）算法有一个主要问题：它允许更多的请求在窗口的边缘通过。

滑动窗口日志算法解决了这个问题。

记录每一个请求的到达时间（timestamp log），每次请求来时清理超过窗口范围的旧记录，计算当前数量是否超限。

它的工作原理如下：

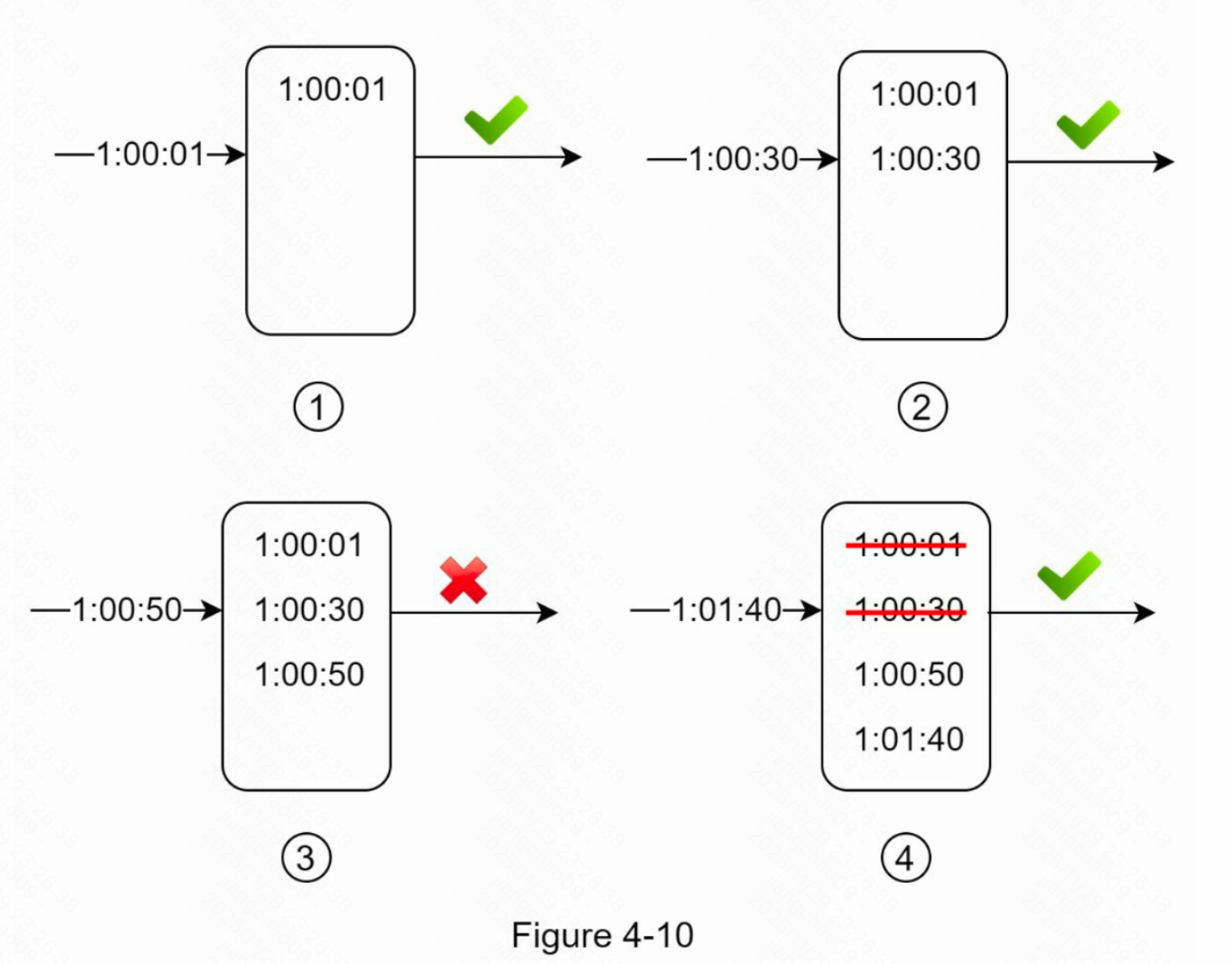
- 该算法对请求的时间戳进行跟踪。时间戳数据通常保存在缓存中，如Redis的sorted[8] 。

- 当一个新的请求进来时，删除所有过期的时间戳。过时的时间戳被定义为比当前时间窗口的开始时间更早的时间戳。

- 将新请求的时间戳添加到日志中

- 如果日志大小与允许的计数相同或更低，则接受请求。否则，它将被拒绝。

我们用图4-10所示的一个例子来解释该算法。



在这个例子中，限流器允许每分钟2个请求。通常情况下，Linux的时间戳会存储在日志中。然而，在我们的例子中，为了提高可读性，使用了人类可读的时间表示法。

- 当一个新的请求在1:00:01到达时，该日志是空的。因此，该请求被允许。

- 一个新的请求在1:00:30到达，时间戳1:00:30被插入到日志中。插入后，日志大小为2，不大于允许的数量，因此，该请求被允许。

- 一个新的请求在1:00:50到达，时间戳被插入到日志中。插入后，日志大小为3，大于允许的大小2。因此，这个请求被拒绝，尽管时间戳仍然在日志中。

- 一个新的请求在 1:01:40 到达。在[1:00:40,1:01:40]范围内的请求是在最新的时间范围内，但在1:00:40之前发送的请求是过时的。

- 两个过期的时间戳1:00:01和1:00:30被从日志中删除。在删除操作之后，日志大小变成了2；因此，请求被接受。

#### 优点

最精确、最平滑的限流方式

理想的理论限流方案

这种算法实现的速率限制是非常准确的。在任何滚动窗口中，请求都不会超过速率限制。

#### 缺点

存储成本高（需要存储每一次请求）

删除过期记录成本高

高并发下日志维护负担大

该算法消耗了大量的内存，因为即使一个请求被拒绝，其时间戳仍可能被存储在内存中

#### 适用场景

小流量业务需要极准确限流

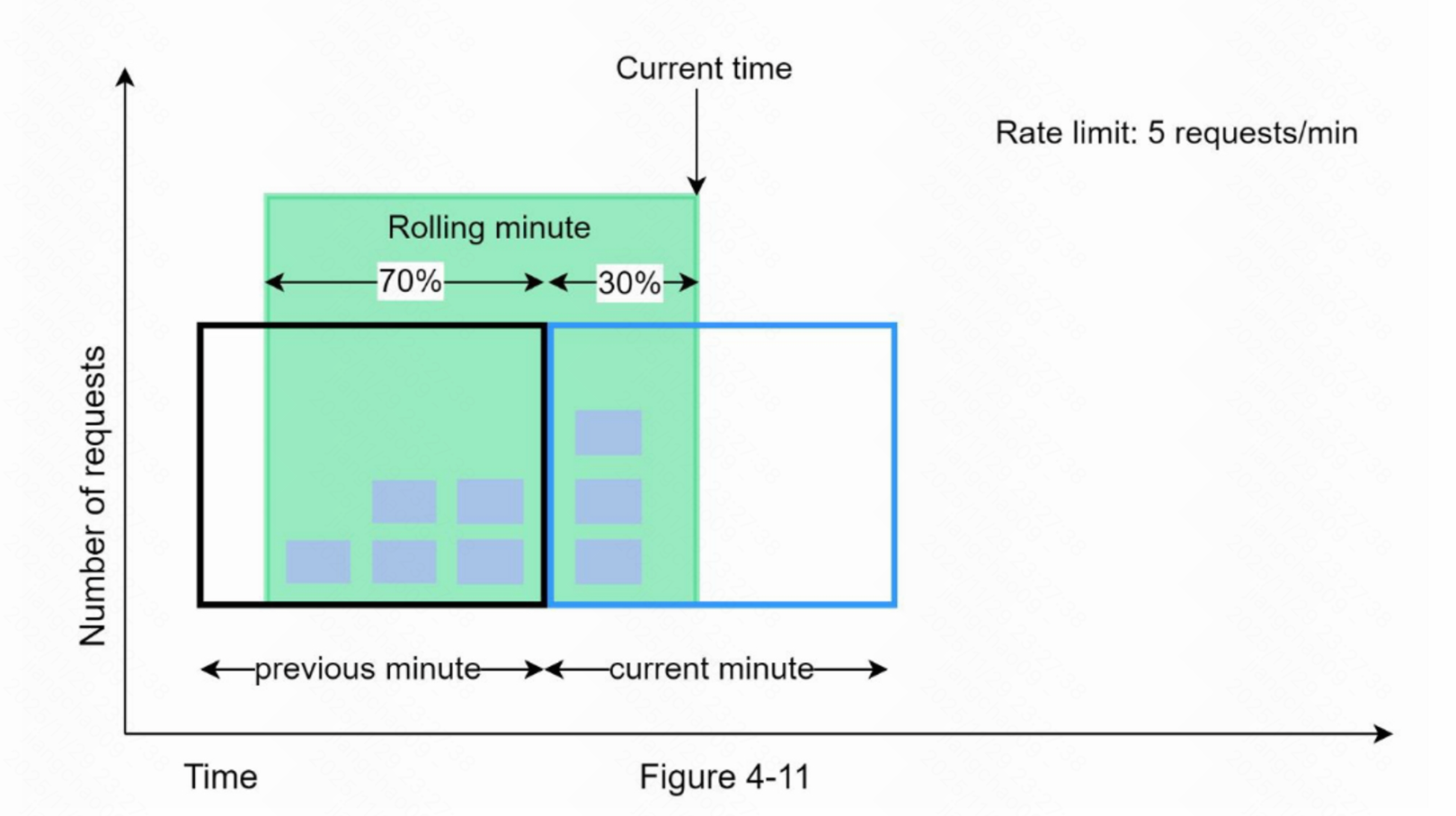
某些安全风控场景（如登录次数限制）

### 滑动窗口计数器算法（Sliding Window Counter）

#### 原理

滑动窗口计数器（Sliding window counter）算法是一种混合方法，结合了固定窗口计数器和滑动窗口日志。该算法可以通过两种不同的方法来实现。我们将在本节中解释一种实现方法，并在本节末尾提供另一种实现方法的参考。

图4-11说明了这种算法的工作原理：



假设限流器允许每分钟最多有7个请求，在上一分钟有5个请求，当前一分钟有3个请求。对于在当前分钟内到达30%位置的新请求，滚动窗口中的请求数用以下公式计算：

- 当前窗口中的请求数量 + 上一个窗口中的请求数量 \* 滚动窗口和上一个窗口的重叠百分比

- 使用这个公式，我们得到 3 + 5 \* 0.7 = 6.5个请求。根据不同的使用情况，这个数字可以向上或向下取整。在我们的例子中，它被向下四舍五入为6。

由于限流器每分钟最多允许7个请求，当前的请求可以通过。然而，再收到一个请求后，就会达到限制。

由于篇幅所限，我们在此不讨论其他的实现。有兴趣的读者可以参考参考资料[9]。这种算法并不完美。它有优点也有缺点。

#### 区别

滑动窗口日志（Sliding Log） = 记录每个请求的时间戳 → 精确到每一个请求

滑动窗口计数器（Sliding Window Counter） = 把时间分成小格子(buckets) → 统计每个 bucket 的计数

滑动日志精确，但很慢。

滑动计数器不够精确，但非常高效。



#### 优点

解决了固定窗口突然双倍峰值的问题

限流更平滑：它平滑了流量的峰值，因为速率是基于前一个窗口的平均速率。

内存高效

#### 缺点

实现较复杂（需维护多个 bucket）

存储和计算成本略高

它只适用于不太严格的回看窗口。它是实际速率的近似值，因为它假设前一个窗口的请求是均匀分布的。然而，这个问题可能并不像它看起来那么糟糕。根据Cloudflare[10]所做的实验，在4亿个请求中，只有0.003%的请求被错误地允许或限制速率。

#### 适用场景

API 网关

对突发流量敏感的系统

### 基于队列长度/延迟的自适应限流（如 TCP BBR/Hystrix）

#### 原理

根据系统当前的：

- 平均延迟

- 错误率

- 队列长度

动态调整限流阈值。

例如 Hystrix 会：

- 延迟升高 → 自动降低阈值（限流）

- 延迟降低 → 放宽限流

#### 优点

完全根据系统压力自适应

防止被打垮（熔断 + 限流组合）

响应实时动态

#### 缺点

实现复杂

调参困难

#### 适用场景

大规模微服务（Netflix Hystrix/Resilience4j）

要求稳定性的系统（支付、订单）

### 高层次的架构

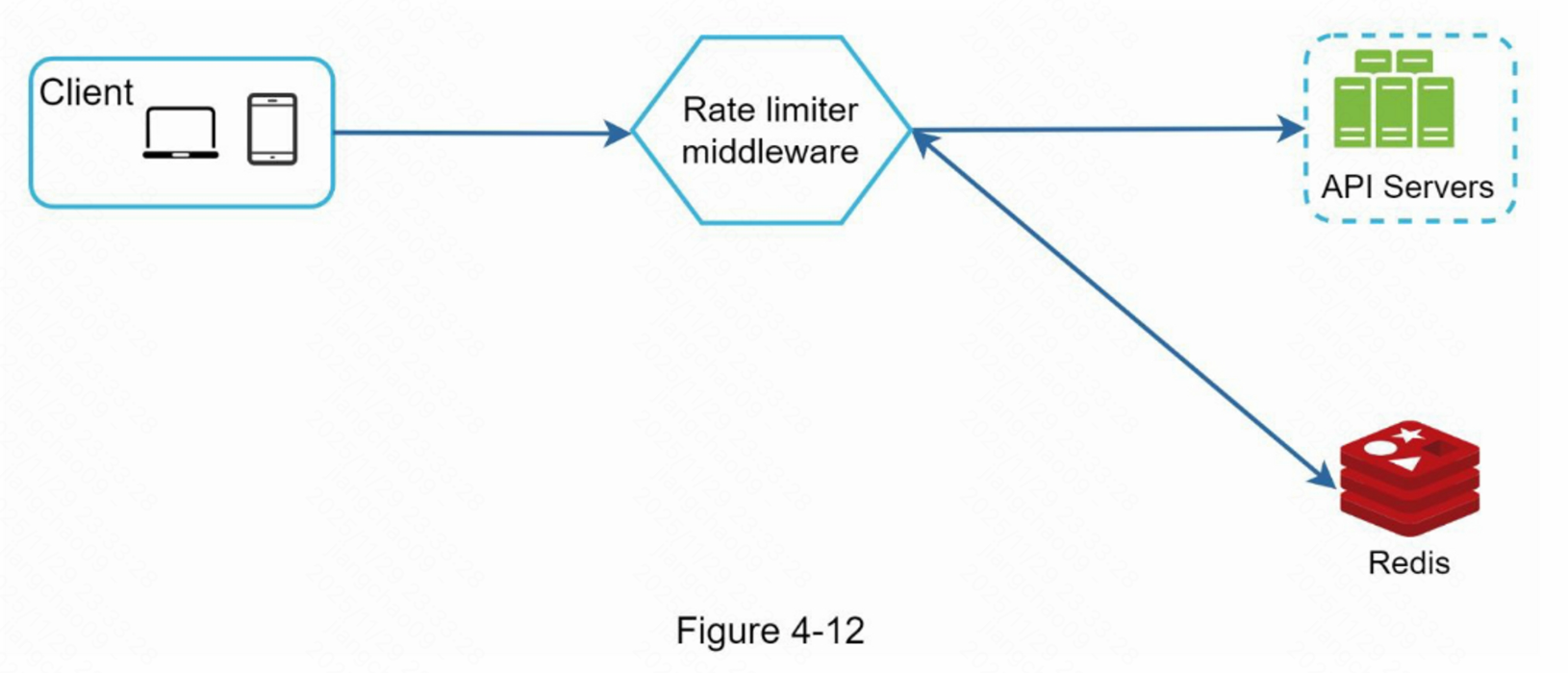
限流算法的基本思想很简单。在高层次上，我们需要一个计数器来跟踪来自同一用户、IP地址等的多少个请求。如果计数器大于限制值，则请求被禁止。

我们应该在哪里存储计数器？由于磁盘访问速度慢，使用数据库并不是一个好主意。选择内存缓存是因为它速度快并且支持基于时间的过期策略。 例如，Redis[11]是实现速率限制的一个流行选择。它是一个内存中的存储，提供两个命令：INCR和EXPIRE。

INCR：它使存储的计数器加1。

EXPIRE：它为计数器设置一个超时。如果超时过后，计数器会被自动删除。

图4-12显示了速率限制的高层结构，其工作原理如下：



- 客户端向限流中间件发送请求

- 限流中间件从Redis中相应的桶中获取计数器，并检查是否达到限制

如果达到限制，则拒绝该请求

如果没有达到限制，请求会被发送到API服务器。同时，系统会增加计数器并将其保存回Redis。

# 深入设计

图4-12中的高层设计并没有回答以下问题：

如何创建速率限制规则？这些规则储存在哪里？

如何处理受限的请求？

在这一节中，我们将首先回答关于限流规则的问题，然后介绍处理限流请求的策略。最后，我们将讨论分布式环境中的限流、详细的设计、性能优化和监控。

## 限流规则

Lyft开源了他们的速率限制组件[12]。我们将窥探该组件的内部情况，并看看一些速率限制规则的例子。

domain: messaging

descriptors:

- key: message\_type

Value: marketing

rate\_limit:

unit: day

requests\_per\_unit: 5

在上述例子中，系统被配置为每天最多允许5条营销信息。下面是另一个例子：

domain: auth

descriptors:

- key: auth\_type

Value: login

rate\_limit:

unit: minute

requests\_per\_unit: 5

这个规则显示，客户不允许在1分钟内登录超过5次。规则一般写在配置文件中并保存在磁盘上。

## 超过速率限制

如果一个请求被限制了速率，API会向客户端返回一个HTTP响应代码429（请求太多）。根据不同的使用情况，我们可能会将速率受限的请求排队等候以后处理。例如，如果一些订单由于系统过载而受到速率限制，我们可以保留这些订单以便以后处理。

## 限流器请求头

一个客户如何知道它是否被节流？客户端如何知道在被节流之前允许的剩余请求的数量？答案就在HTTP响应头中。限流器向客户端返回以下 HTTP 标头：

X-Ratelimit-Remaining：窗口内允许请求的剩余数量

X-Ratelimit-limit：它表示客户端在每个时间窗口可以进行多少次调用

X-Ratelimit-Retry-After：等待的秒数，直到你可以再次提出请求而不被节流。

当用户发送过多请求时，将向客户端返回 429 too many requests 错误和 X-Ratelimit-Retry-After 标头。

## 详细设计

图 4-13 给出了系统的详细设计。



规则被存储在磁盘上。工作者经常从磁盘中提取规则，并将其存储在高速缓存中。

当客户端向服务器发送请求时，该请求首先被发送到限流中间件。

限流中间件从缓存中加载规则。它从Redis缓存中获取计数器和最后一次请求的时间戳。根据响应，限流器决定：

如果请求没有速率限制，它将被转发到API服务器。

如果请求受到速率限制，限流器会向客户端返回 429 too many requests 错误。同时，请求被丢弃或转发到队列。

# 分布式环境下的限流器

构建一个在单个服务器环境下运行的限流器并不困难。然而，将系统扩展到支持多个服务器和并发线程是另一回事。这里有两个挑战：

竞争条件

同步问题

## 竞争条件

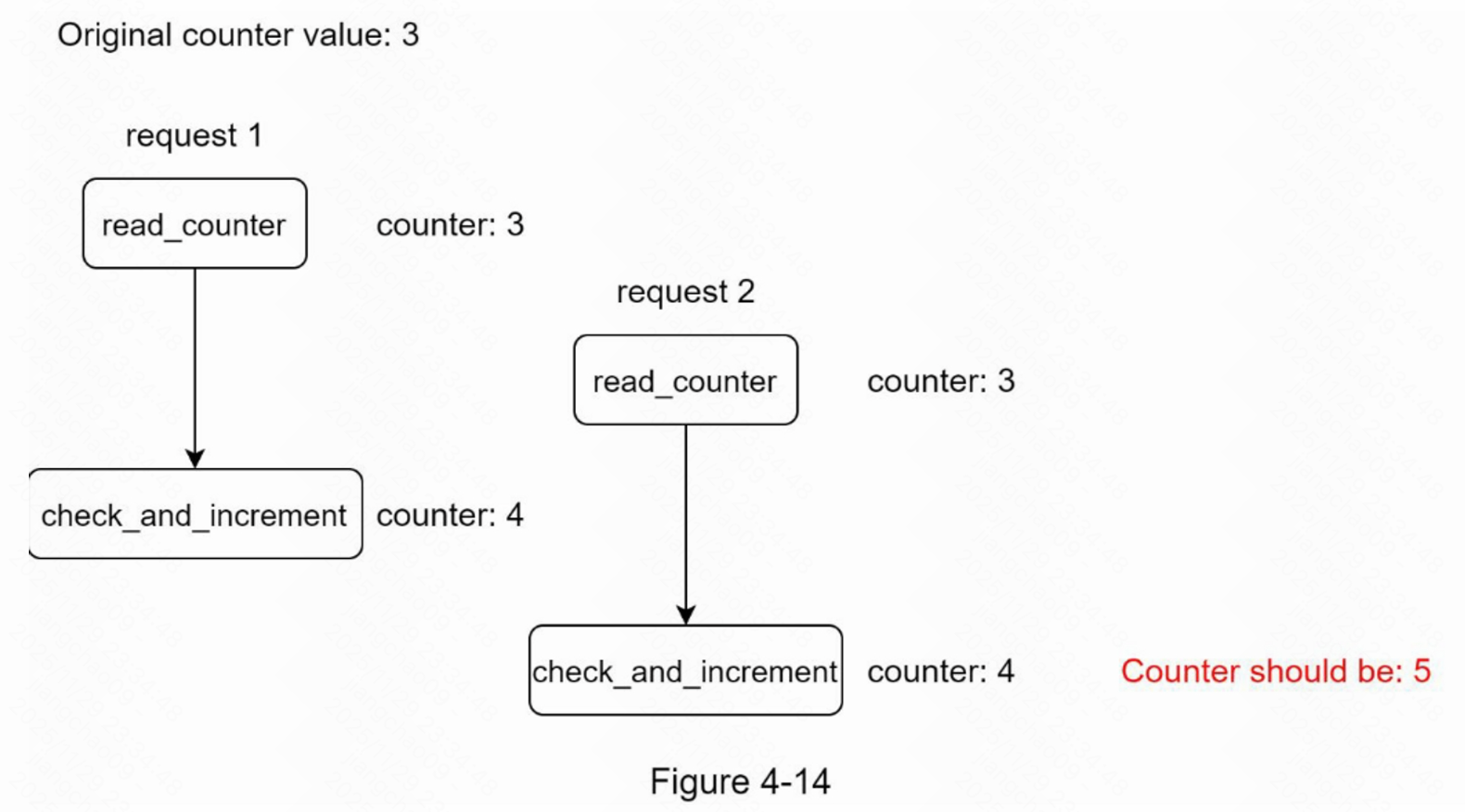
如前所述，限流器在高层的工作原理如下

- 从Redis读取计数器的值

- 检查 ( 计数器 + 1 ) 是否超过阈值

- 如果不是，则将 Redis 中的计数器值加 1

如图4-14所示，在高度并发的环境中会发生竞争条件。

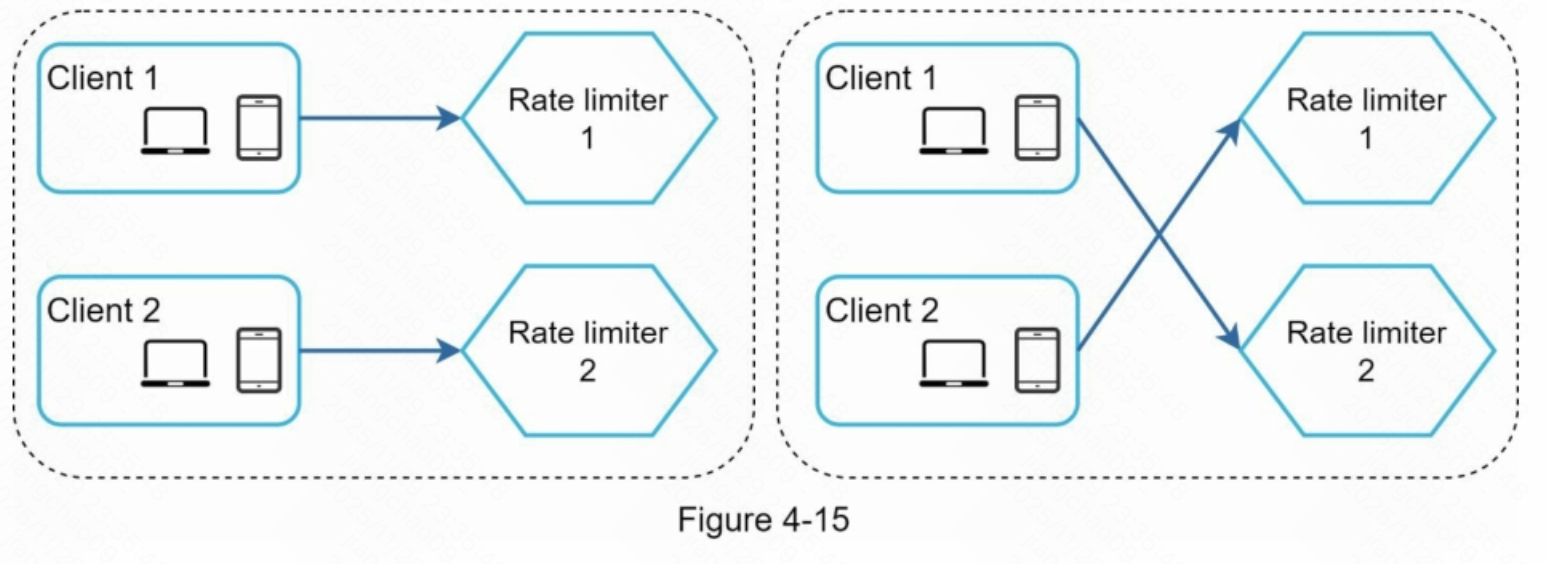


假设 Redis 中的计数器值为 3。如果两个请求在其中一个请求写回值之前同时读取计数器值，则每个请求都会将计数器加 1 并在不检查另一个线程的情况下将其写回。 两个请求（线程）都认为它们具有正确的计数器值 4。但是，正确的计数器值应该是 5。

锁是解决竞争条件最明显的解决方案。 但是，锁会显着降低系统速度。 通常使用两种策略来解决这个问题：Lua 脚本[13]和 Redis [8]中的 sorted sets 数据结构。对这些策略感兴趣的读者可以参考相应的参考资料[8] [13]。

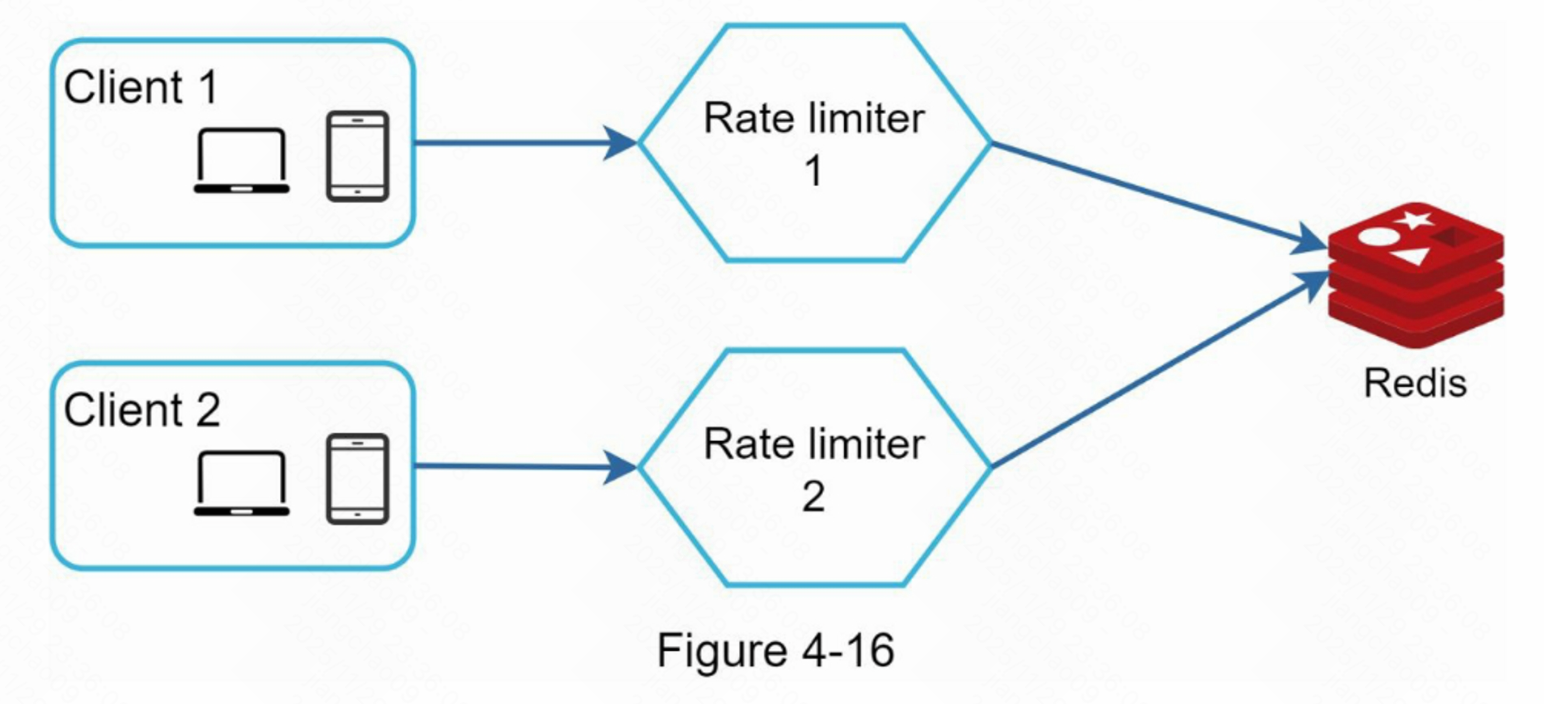
## 同步问题

同步是分布式环境中要考虑的另一个重要因素。要支持数百万用户，一个限流器服务器可能不足以处理流量。当使用多个限流器服务器时，需要同步。 例如，在图 4-15 的左侧，客户端 1 将请求发送到限流器 1，客户端 2 将请求发送到限流器 2。由于 Web 层是无状态的，客户端可以将请求发送到不同的限流器，如图所示，在图 4-15 的右侧。如果没有发生同步，限流器 1 不包含有关客户端 2 的任何数据。因此，限流器无法正常工作。



一种可能的解决方案是使用粘性会话，允许客户端将流量发送到相同的限流器。这个解决方案是不可取的，因为它既不可扩展也不灵活。更好的方法是使用像 Redis 这样的集中式数据存储。

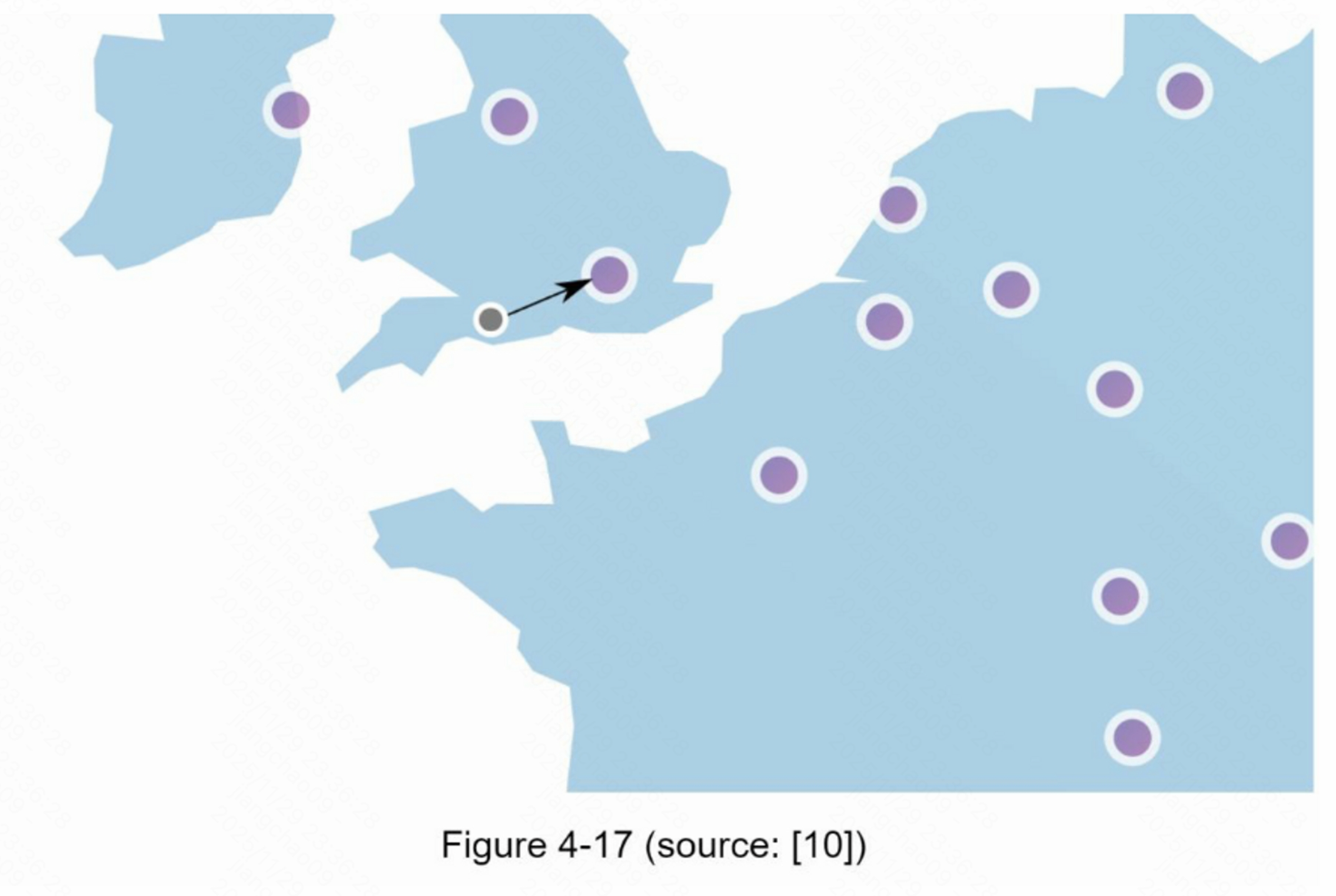
设计如图 4-16 所示：



## 性能优化

性能优化是系统设计面试中的一个常见话题。我们将涉及两个方面的改进。

首先，多数据中心的设置对限流器至关重要，因为对于远离数据中心的用户来说，延迟很高。大多数云服务提供商在世界各地建立了许多边缘服务器位置。例如，截至2020年5月20日，Cloudflare有194个地理上分布的边缘服务器[14]。流量被自动路由到最近的边缘服务器，以减少延时。



第二，用最终的一致性模型来同步数据。

## 监控

限流器到位后，最重要的是要收集分析数据以检查限流器是否有效。

首先，我们要确保：

限流算法是有效的

限流规则是有效的

例如，如果速率限制规则过于严格，则会丢弃许多有效请求。 在这种情况下，我们想稍微放宽规则。 在另一个示例中，我们注意到当流量突然增加时（如抢购），我们的限流器变得无效。 在这种场景下，我们可能会更换算法来支持突发流量。 令牌桶很适合这种场景。

# 总结

讨论的算法包括：

- 令牌桶

- 漏桶

- 固定窗口计数器

- 滑动窗口日志

- 滑动窗口计数器

然后，我们讨论了系统架构、分布式环境中的限流器、性能优化和监控。 与任何系统设计面试问题类似，如果时间允许，您可以提及其他谈话要点:

- 硬件与软件限流对比

硬件：请求的数量不能超过阈值

软件：请求可以在短时间内超过阈值

- 不同级别的限流。在本章中，我们只讨论了应用程序级别（HTTP：第 7 层）的限流，可以在其他层应用限流。例如，你可以使用Iptables[15]（IP：第3层）按IP地址应用速率限制。 注意：开放系统互连模型（OSI模型）有7层[16]。第1层：物理层，第2层：数据链路层，第3层：网络层，第4层：传输层，第5层：会话层，第6层：表示层，第7层：应用层。

- 避免被限速，使用最佳实践设计你的客户端：

使用客户端缓存，以避免频繁调用API

了解限制，不要在短时间内发送太多请求

包含捕获异常或错误的代码，以便客户端可以从异常中恢复正常

为重试逻辑增加足够的回退时间

# 参考资料

[1] Rate-limiting strategies and techniques: https://cloud.google.com/solutions/rate-limiting-strategies-techniques

[2] Twitter rate limits: https://developer.twitter.com/en/docs/basics/rate-limits

[3] Google docs usage limits: https://developers.google.com/docs/api/limits

[4] IBM microservices: https://www.ibm.com/cloud/learn/microservices

[5] Throttle API requests for better throughput:

https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html

[6] Stripe rate limiters: https://stripe.com/blog/rate-limiters

[7] Shopify REST Admin API rate limits: https://help.shopify.com/en/api/reference/rest-admin-api-rate-limits

[8] Better Rate Limiting With Redis Sorted Sets: https://engineering.classdojo.com/blog/2015/02/06/rolling-rate-limiter/

[9] System Design — Rate limiter and Data modelling: https://medium.com/@saisandeepmopuri/system-design-rate-limiter-and-data-modelling-9304b0d18250

[10] How we built rate limiting capable of scaling to millions of domains: https://blog.cloudflare.com/counting-things-a-lot-of-different-things/

[11] Redis website: https://redis.io/

[12] Lyft rate limiting: https://github.com/lyft/ratelimit

[13] Scaling your API with rate limiters: https://gist.github.com/ptarjan/e38f45f2dfe601419ca3af937fff574d#request-rate-limiter

[14] What is edge computing: https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/

[15] Rate Limit Requests with Iptables: https://blog.programster.org/rate-limit-requests-with-iptables

[16] OSI model: https://en.wikipedia.org/wiki/OSI\_model#Layer\_architecture