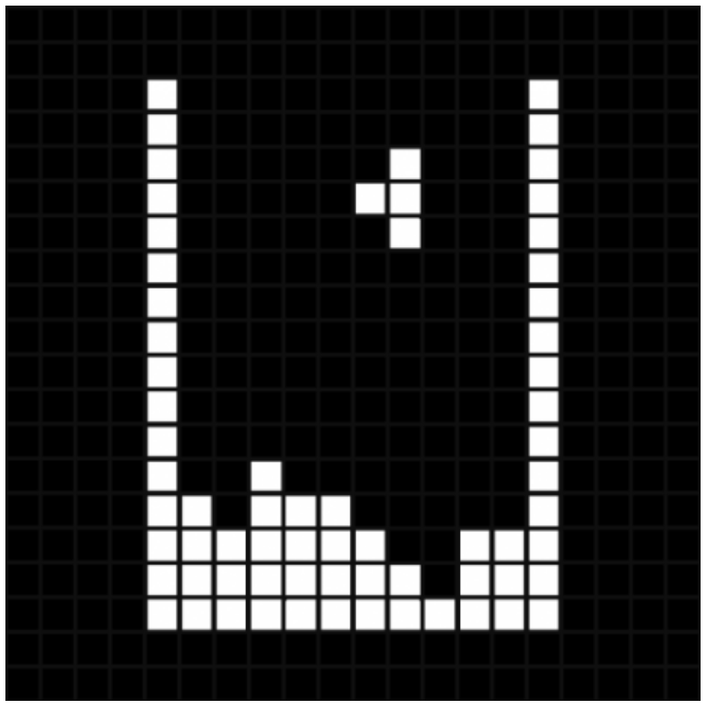
OpenResty 动态流控的几种姿势

OpenResty Open Talk 2019 深圳站

张聪 (@timebug)

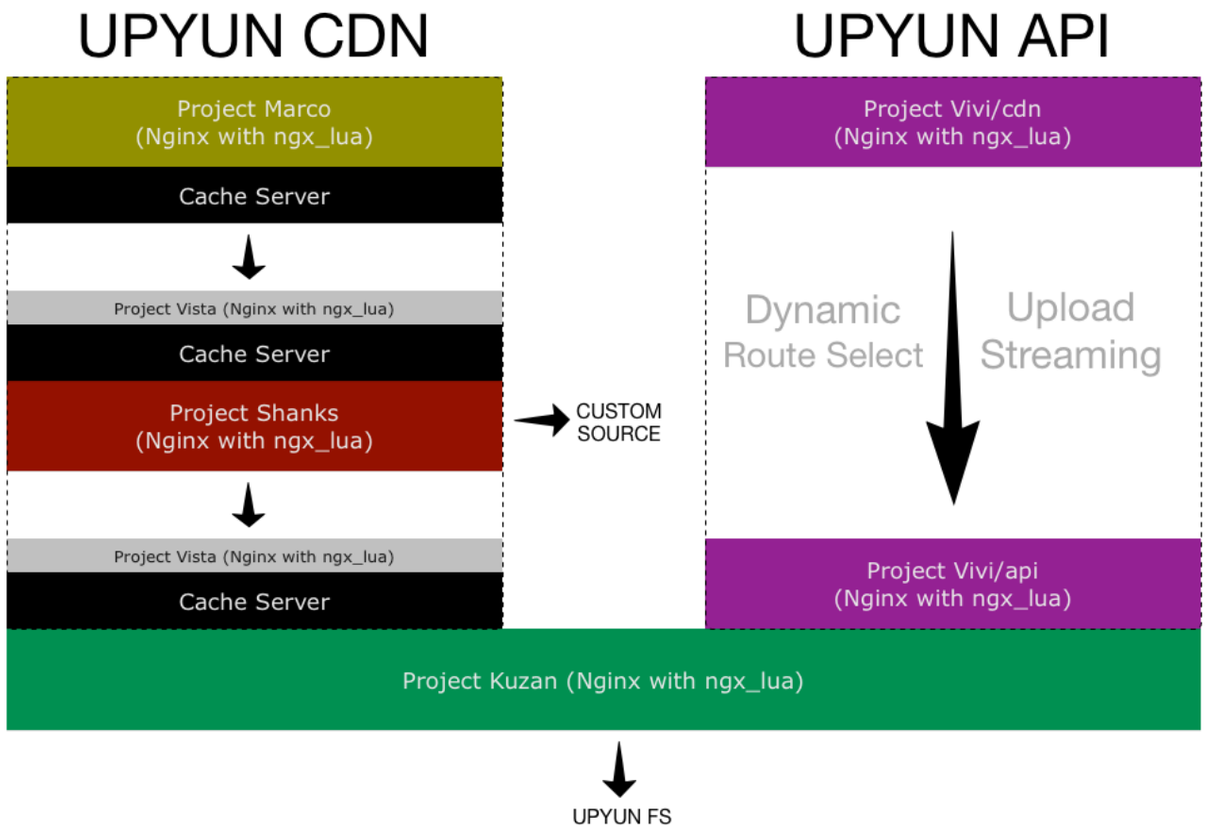


A Systems Engineer at 又拍云

★ Email: timebug.info@gmail.com

★ Github: https://github.com/timebug

https://github.com/upyun/upyun-resty



UPYUN CDN & API is built on top of NGINX with ngx_lua/OpenResty

- What is UPYUN
- What is OpenResty
- Tech Talks
  - 201411 Beijing OSC
  - 201511 Beijing OpenResty Con
  - 201608 Guangzhou UPYUN OpenTalk
  - 201612 Shenzhen OpenResty Con
  - 201712 Hangzhou OpenResty Meetup
  - 201811 Hangzhou OpenResty Con
- Projects
  - Middleware
    - Project Slardar
  - Nginx Modules
    - base64-nginx-module
    - zstd-nginx-module
  - Libraries
    - checkups
    - httpipe
    - redis-ratelimit
    - 17monip
    - sync
    - ctxdump
    - consul
    - load
    - requests
    - limit-rate
  - Community Contributions
    - slice filter and If-Range requests
    - autoindex module and request body
    - ngx.req.raw_header with single line break
    - ngx_lua and filter finalize problem
    - segmentation fault might occur when SSL renegotiation happens
- Work at UPYUN

什么是流控

我个人的理解是（**针对应用层**）：

- 流控通常意义下是通过一些合理的技术手段对入口请求或流量进行有效地疏导和控制，从而使得有限资源的上游服务和整个系统能始终在健康的设计负荷下工作，同时在不影响绝大多数用户体验的情况下确保“**利益**”最大化。

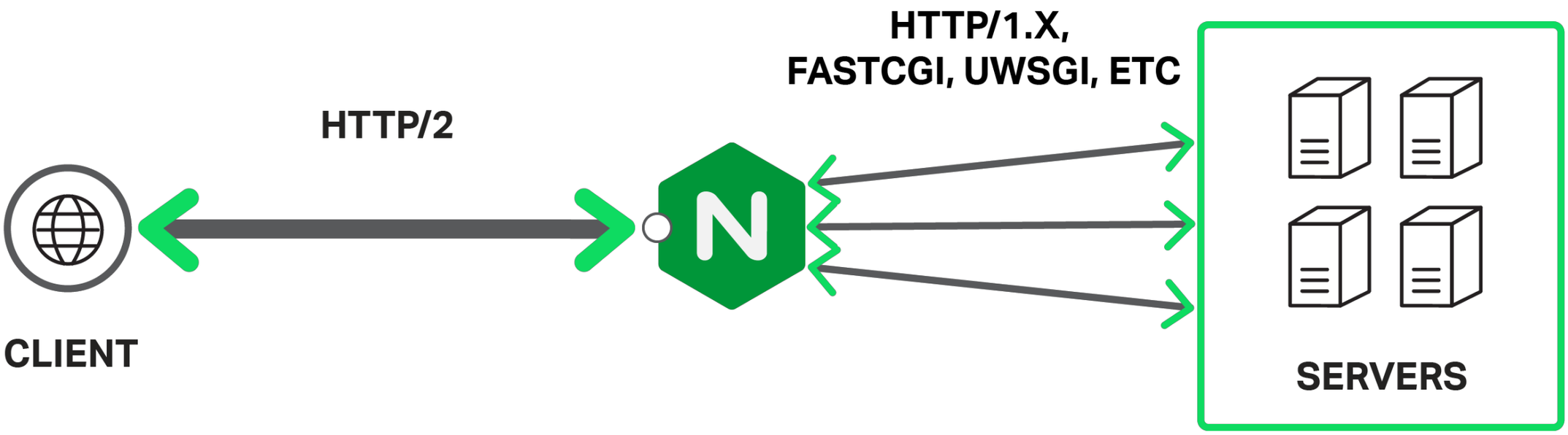
- 流控有时候也是在考虑**安全**和**成本**时的一个手段。

为什么要流控

- 为了**业务数据安全**，针对关键密码认证请求进行有限次数限制，避免他人通过字典攻击暴力破解。
- 在保障正常用户请求频率的同时，限制**非正常速率**的恶意 **DDoS** 攻击请求，拒绝非人类访问。
- 控制上游应用在**同一时刻处理的用户请求**数量，以免出现并发资源竞争导致体验下降。
- 上游业务处理能力有限，如果某一时刻**累计未完成任务超过设计最大容量**，会导致整体系统出现不稳定甚至持续恶化，需要时刻保持在安全负荷下工作。
- 集群模式下，负载均衡也是流控最基础的一个环节，当然也有一些业务无法精确进行前置负载均衡，例如图片处理等场景就容易出现单点资源瓶颈，此时需要根据上游节点**实时负载情况**进行主动调度。
- 在实际的业务运营中，往往处于**成本考虑**，还需要进行流量整形和带宽控制，包括下载限速和上传限速，以及在特定领域例如终端设备音视频播放场景下，根据实际码率进行针对性速率限制等。
- …（当然，还有很多上面没提到的原因）

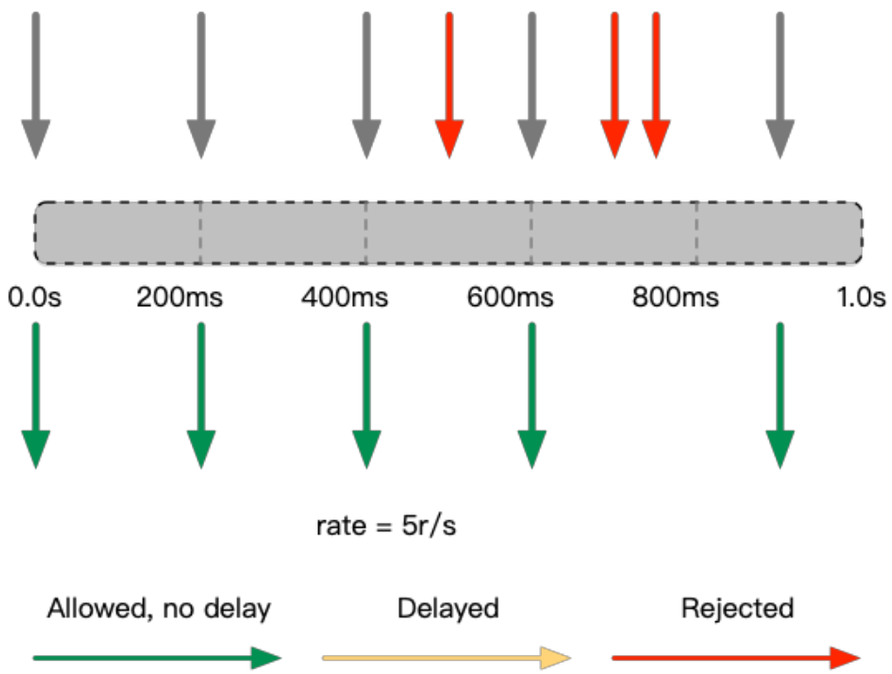
怎么做流控

- 经典的 NGINX 方式。
- 灵活的 **OpenResty** 方式。
- 又拍云的线上实际案例。


HTTP/1.X,
FASTCGI, UWSGI, ETC
HTTP/2
CLIENT
SERVERS

NGINX 请求速率限制（1）

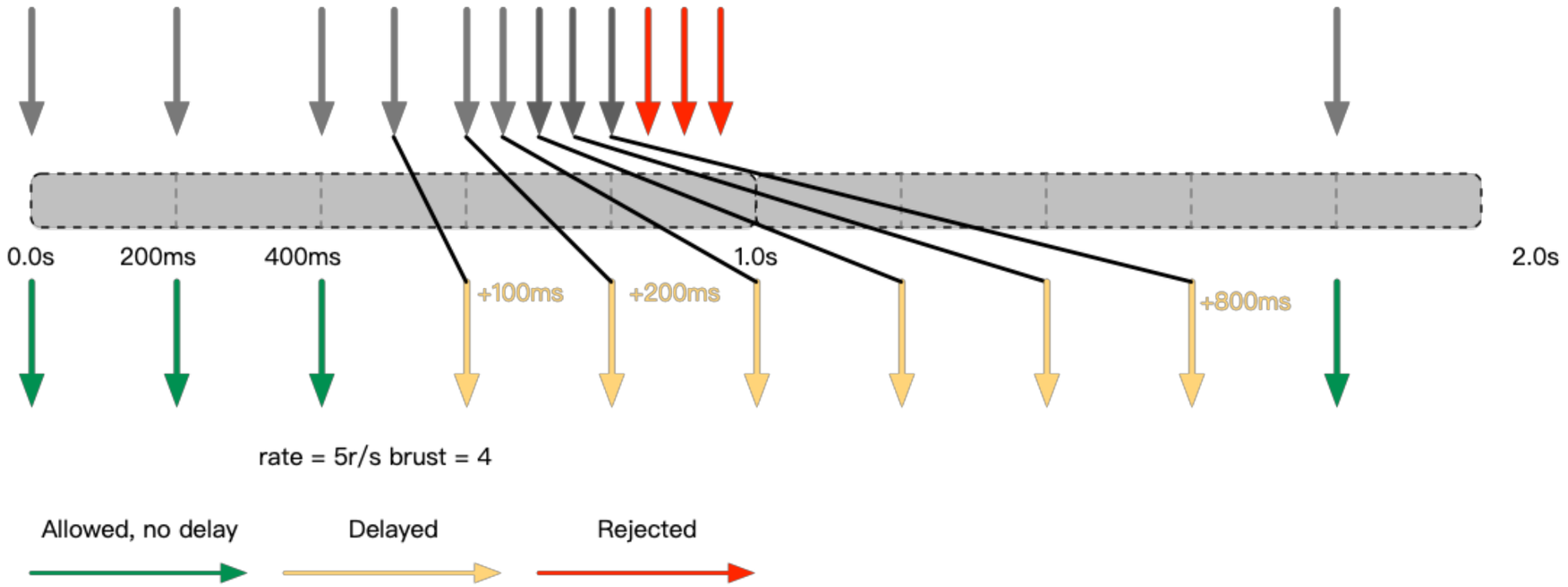
```
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=5r/s;

limit_req_log_level error;
limit_req_status 429;

server {
    location /upyun/ {
        limit_req zone=mylimit;

        proxy_pass http://website;
    }
}
```



➡ **ngx_http_limit_req_module**

- 针对来源 IP，限制其请求速率为 **5r/s**。
- 意味着，相邻请求间隔至少 200ms，否则拒绝。
- 但实际业务中，偶尔有些徒增也是正常的。

NGINX 请求速率限制（2）

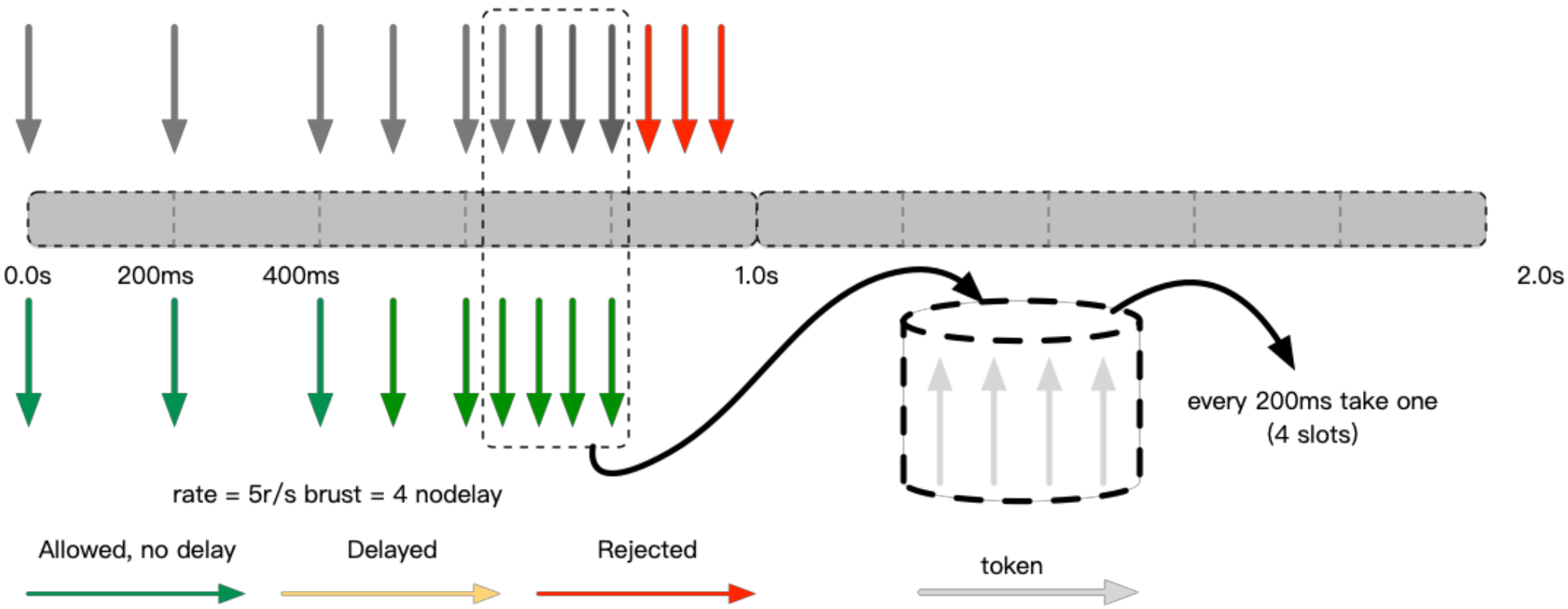


- 设置 **brust = 4**，表示在超过限制速率 **5r/s** 的时候，同时最多允许额外有 4 个请求排队等候，待平均速率回归正常后，队列最前面的请求会优先被处理。

- 在已经存在 4 个请求同时等候的情况下，此时“立刻”过来的请求就会被拒绝。

- 虽然允许了一定程度的突发，但有些业务场景下，排队导致的请求延迟增加是不可接受的，例如上图中突发队列队尾的那个请求被滞后了 800ms 才进行处理。

NGINX 请求速率限制（3）

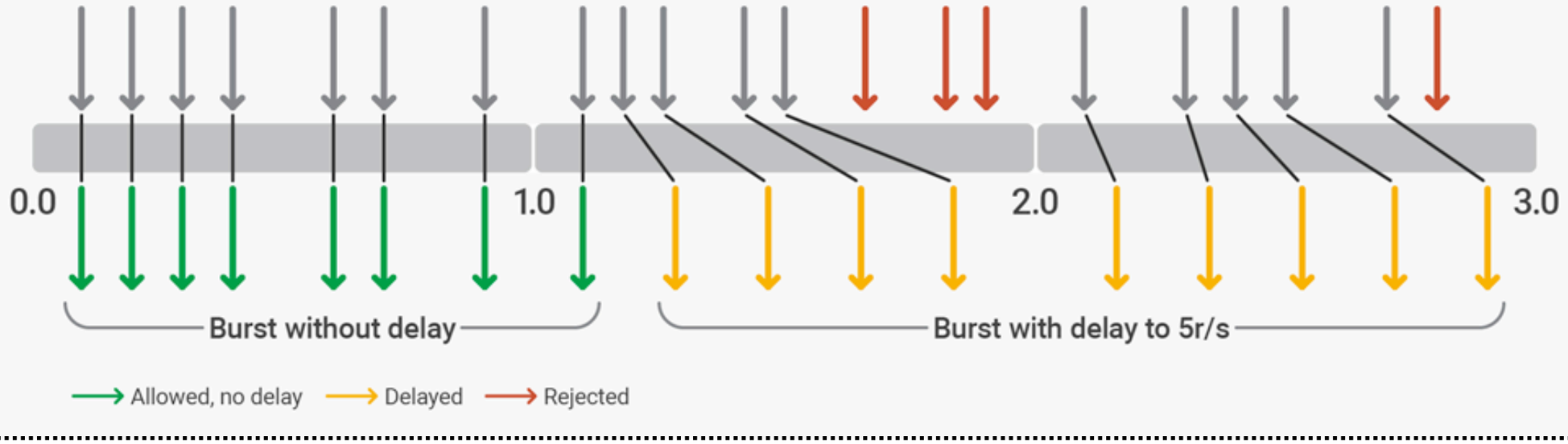


- nodelay 参数配合 brust = 4 就可以使得突发时需要等待的请求立即得到处理，与此同时，模拟一个插槽个数为 4 的“令牌”队列（桶）。

- 从抽象的角度描述下这个过程，该“令牌”桶会每隔 200ms 释放一个“令牌”，空出的槽位等待新的“令牌”进来，若桶槽位被填满，随后突发的请求就会被拒绝。

- 这个模式下，在控制请求速率的同时，允许了一定程度的突发，并且尽可能改善了延迟体验。

NGINX 请求速率限制（4）



- NGINX 1.15.7 还支持 **delay=number** 和 **brust=number** 参数配合使用。

- 在有些特定场景下，我们既需要保障正常的少量关联资源能够快速地加载，同时也需要对于突发请求及时地进行限制，而 **delay** 参数能更精细地来控制这类限制效果。

- 例如某页面引用的资源大概 4 ~ 6 个，需要确保并发加载这些资源，但又肯定不会超过 12 个，那么针对该页面的限制策略除了 **burst=12** 外，还可以设置 **delay=8**，在突发请求的情况下，**其中前 8 个不需要进行延迟处理**，后 4 个则依然排队等候处理。

NGINX 并发连接数限制

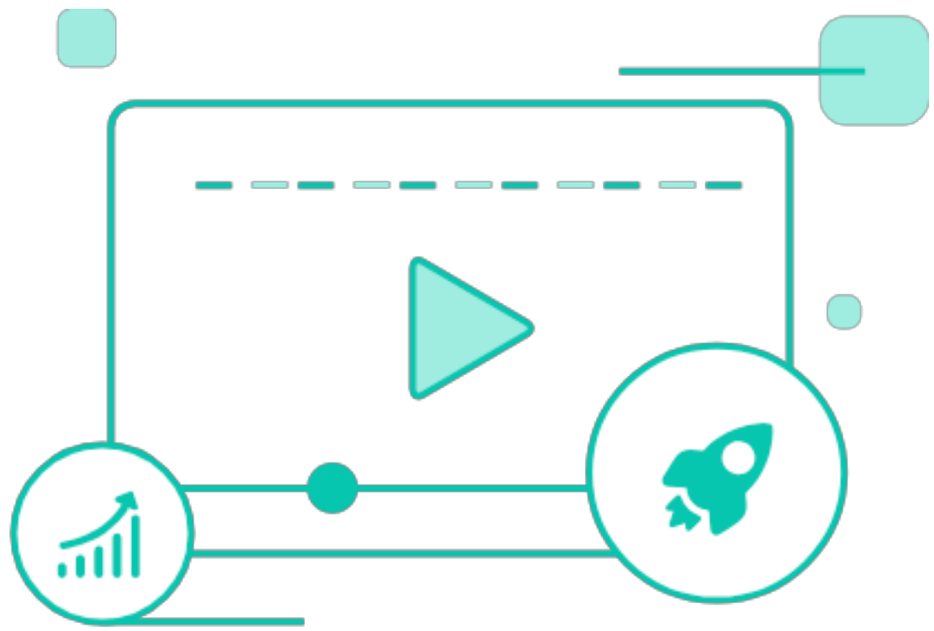
```
limit_conn_zone $binary_remote_addr zone=addr:10m;

limit_conn_log_level error;
limit_conn_status 503;

server {
    location /download/ {
        limit_conn addr 1;
     }
}
```

➡ **ngx_http_limit_conn_module**

- 对于处理中的请求，该模块在读完请求头全部内容后才开始计数。
- 在 HTTP/2 和 SPDY 协议下，每一个并发请求都会当做一个独立的连接。

NGINX 下载带宽限制

```
location /download/ {
    limit_rate_after 500k;
    limit_rate 20k;
}
```



➡ **ngx_http_core_module**

◉ 在下载完前面 500 KB 数据后，对接下来的数据以每秒 20 KB 速度进行限制。

◉ 常常用于一些文件下载、视频播放等业务场景，避免不必要的流量浪费。

OpenResty 动态流控

- 我们需要更加丰富的流控策略!
- 我们需要更加灵活的配置管理!
- 我们需要在 NGINX 请求生命周期的更多阶段进行控制!
- 我们需要跨机器进行状态同步!
- NGINX PPPlus ? NGINX+

OpenResty 动态流控

请求速率限制/并发连接数限制

```
local limit_req = require "resty.limit.req"

local lim, err = limit_req.new("mylimit", 5, 9)

local delay, err = lim:incoming(ngx.var.binary_remote_addr, true)
if not delay then
    if err == "rejected" then
        return ngx.exit(429)
    end
    return ngx.exit(500)
end

if delay >= 0.001 then
    ngx.sleep(delay)
end
```

➡ **lua-resty-limit-traffic (resty.limit.req)**

◉ 上述 Lua 代码基本效果等同于 NGINX **limit_req** 模块 **rate=5r/s brust=4** 的限速配置。

◉ 但 Lua 实现更加灵活以及几乎可以在任意上下文使用。

➡ **lua-resty-limit-traffic (resty.limit.conn)**

◉ 功能和 NGINX **limit_conn** 一致，但 Lua 版本允许突发连接进行短暂延迟等候。

OpenResty 动态流控 - 请求数量限制

```
local limit_count = require "resty.limit.count"

local lim, err = limit_count.new("mylimit", 5000, 3600)

local delay, err = lim:incoming(ngx.req.get_headers()["Authorization"], true)
if not delay then
    if err == "rejected" then
        ngx.header["X-RateLimit-Limit"] = "5000"
        ngx.header["X-RateLimit-Remaining"] = 0
        return ngx.exit(503)
    end
    return ngx.exit(500)
end

ngx.header["X-RateLimit-Limit"] = "5000"
ngx.header["X-RateLimit-Remaining"] = err -- current remaining number
```

➡ **lua-resty-limit-traffic (resty.limit.count)**

- 和 Github API Rate Limiting 接口设计类似。
- 该 Lua 模块用于对单位窗口时间内累计请求数量进行限制。

OpenResty 动态流控 - 跨机器速率限制

```
local ratelimit = require "resty.redis.ratelimit"

local lim, err = ratelimit.new("mylimit", "5r/s", 4, 0)

local red = { host = "127.0.0.1", port = 6379, timeout = 1 }

local delay, err = lim:incoming(ngx.var.binary_remote_addr, red)
if not delay then
    if err == "rejected" then
        return ngx.exit(429)
    end
    return ngx.exit(500)
end

if delay >= 0.001 then
    ngx.sleep(delay)
end
```

➡ **lua-resty-redis-ratelimit (resty.redis.ratelimit)**

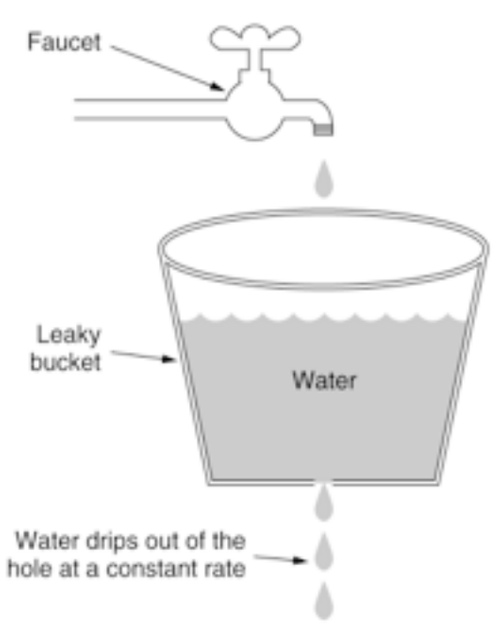
◉ 和 NGINX **limit_req** 以及 **resty.limit.req** 一样，都是基于**漏桶算法**对平均请求速率进行限制。不同的是，该模块将信息保存在 Redis 从而实现多 NGINX 实例状态共享。

◉ 借助于 **Redis Lua Script** 机制，使得跨机器状态变更操作能保证原子性。同时，该模块支持在整个集群层面禁止某个非法用户一段时间，可实现全局自动拉黑功能。

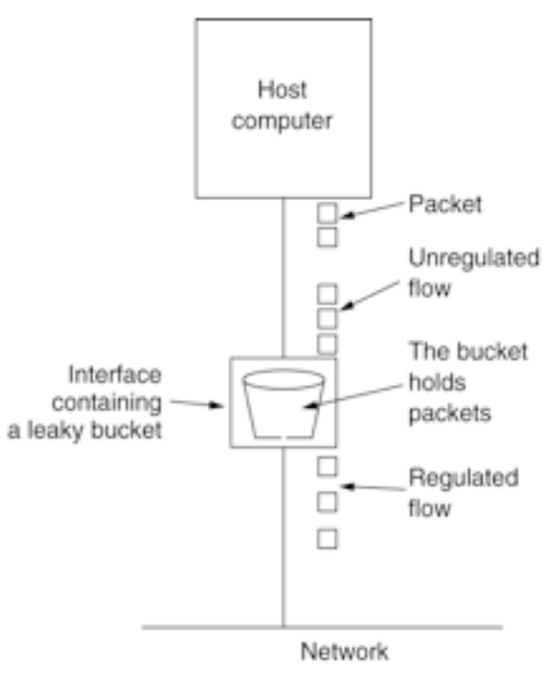
◉ NGINX 和 Redis 交互需要网络 IO，会带来一定延迟开销，仅适合请求量不大，但需要非常精确限制全局请求速率或统计时间跨度非常大的场景。当然这块也可以在牺牲一些精确度和及时性的情况下，在 NGINX 侧进行局部计算。

知识点 - 漏桶算法

```
local ms = math.abs(now - last)
excess = excess - rate * ms / 1000 + 1000

if excess < 0 then
    excess = 0
end

if excess > 0 then
    return BUSY
end
```



(a)



(b)

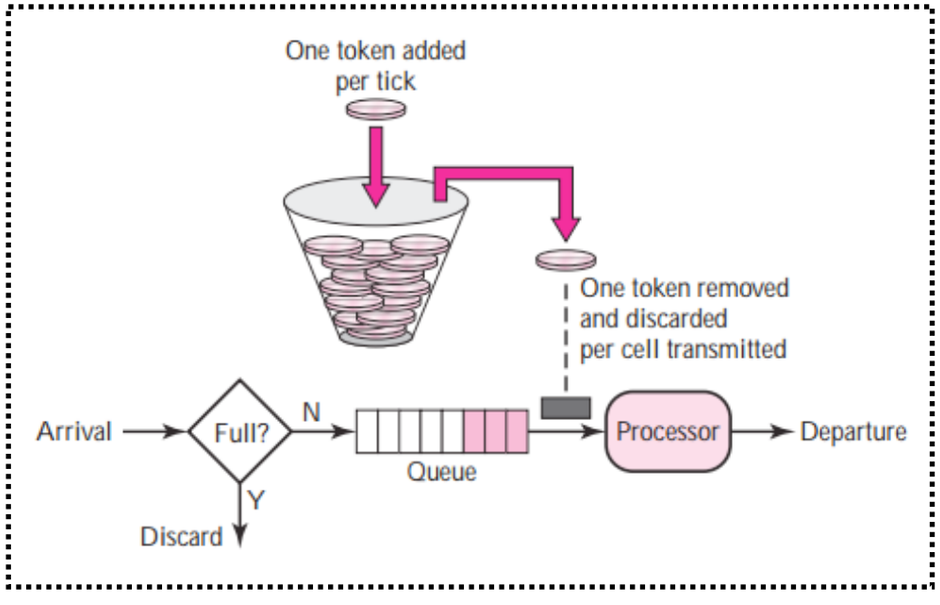
- 在具体实现中，一般最小的速率表示是 0.001r/s，为了直观计算，我们用 1 个水滴（假设单位 t）来表达 0.001 个请求，那么 rate = 5r/s 相当于 5000t/s。
- ms 为当前请求和上一个请求的时间间隔，假设 100ms，单位毫秒，下面公式中除以 1000 等于 0.1s。
- excess 表示上一次超出的水滴数（延迟通过）；特别地，excess < 0 的时候会被重置为 0，此时相当于漏桶处于空桶状态，进来的水滴直接流出了。
- 如果 excess > 0 说明漏桶有水滴堆积了，按我们设计，返回 BUSY，表示繁忙。

OpenResty 动态流控 - 令牌桶限速

```
local lim_global = limit_rate.new("my_limit_rate_store", 100, 6000, 2) -- global 20r/s 6000r/5m
local lim_single = limit_rate.new("my_limit_rate_store", 500, 600, 1) - single 2r/s 600r/5m

local t0, err = lim_global:take_available("__global__", 1)
local t1, err = lim_single:take_available(ngx.var.arg_userid, 1)

if t0 == 1 then
    return -- global bucket is not hungry
else
    if t1 == 1 then
        return -- single bucket is not hungry
    else
        return ngx.exit(503)
    end
end
```

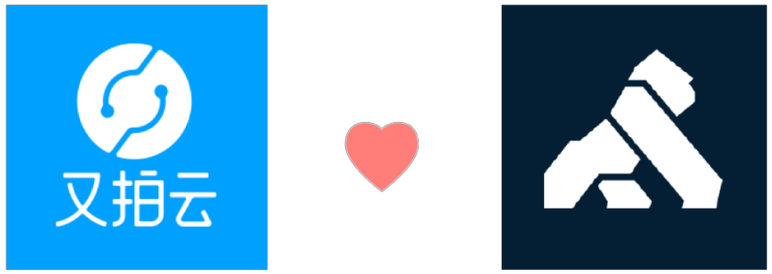
➡ **lua-resty-limit-rate (resty.limit.rate)**

- 该 Lua 模块基于令牌桶算法实现，也是流控中的一种经典思想。
- 相比 limit.req 基于漏桶的设计，该模块更加关注容量变化而非相邻请求间的速率，适合有一定弹性设计容量的系统，在资源足够的情况下，速率允许有较大的波动。
- 相比 limit.count 对单位窗口时间内累计请求数量进行限制，该模块在特定配置下，也能达到类似效果，并且能避免在单位时间窗口切换瞬间导致可能双倍的限制请求情况出现。
- 除了请求速率限制（一个令牌一个请求），还能够对字节传输进行流量整形，此时，一个令牌相当于一个字节。

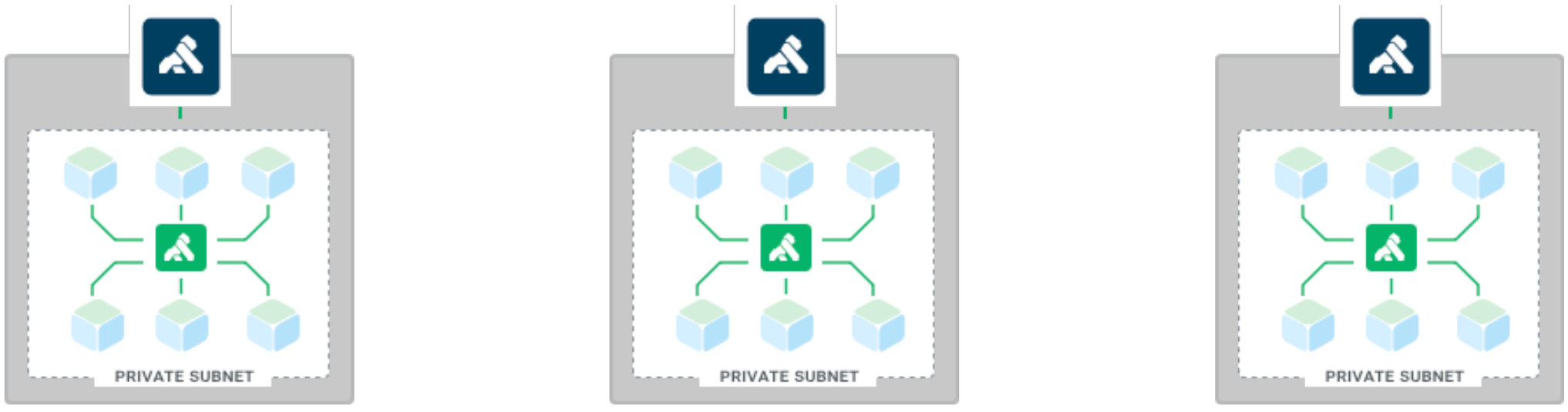


又拍云的线上实际案例

- 海外代理进行上传流量整形，避免跑满传输线路带宽。
- 某 API 请求基于令牌桶针对不同账户进行请求速率控制。
- CDN 特性：IP 访问限制，支持阶梯策略升级。
- CDN 特性：码率适配限速。



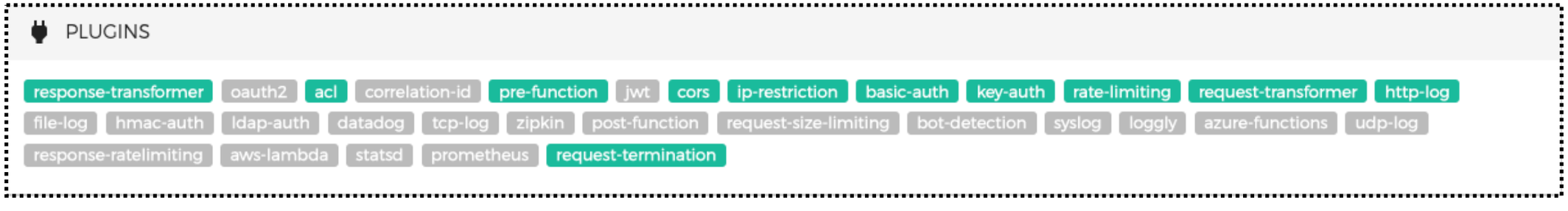
All Web/API services use Kong as the gateway
(Kong is a very famous OpenResty application)



KONG-S
内部站点入口

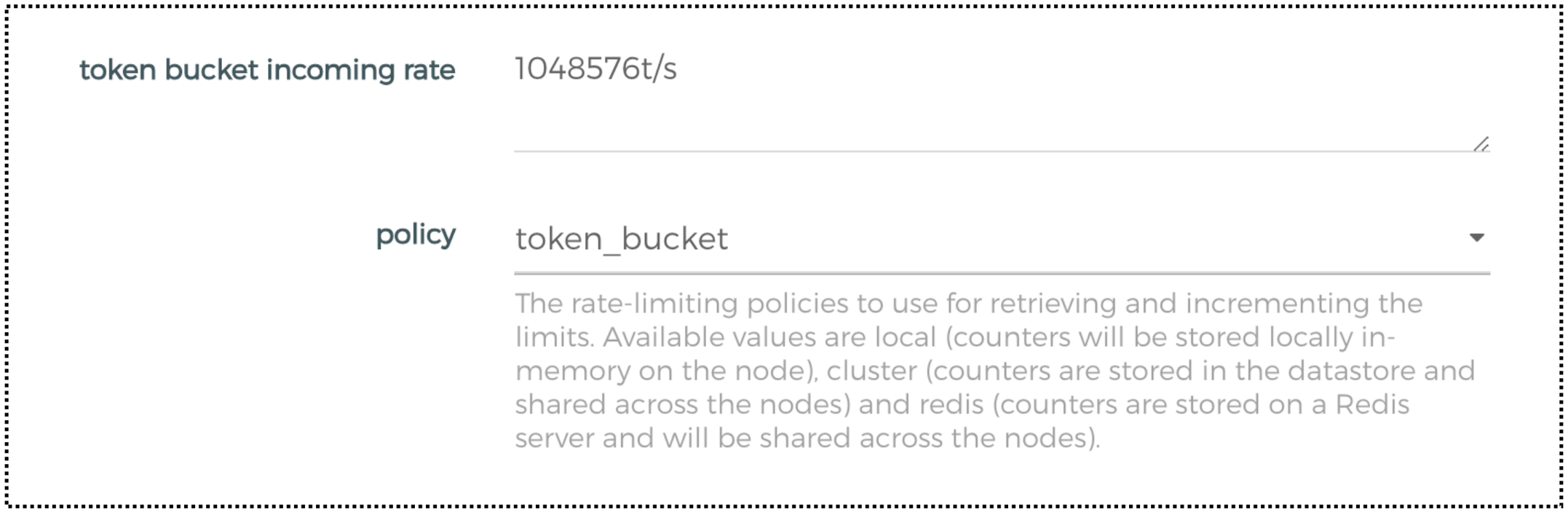
KONG-X
外部站点/API 入口

KONG-HK
海外加速透明转发



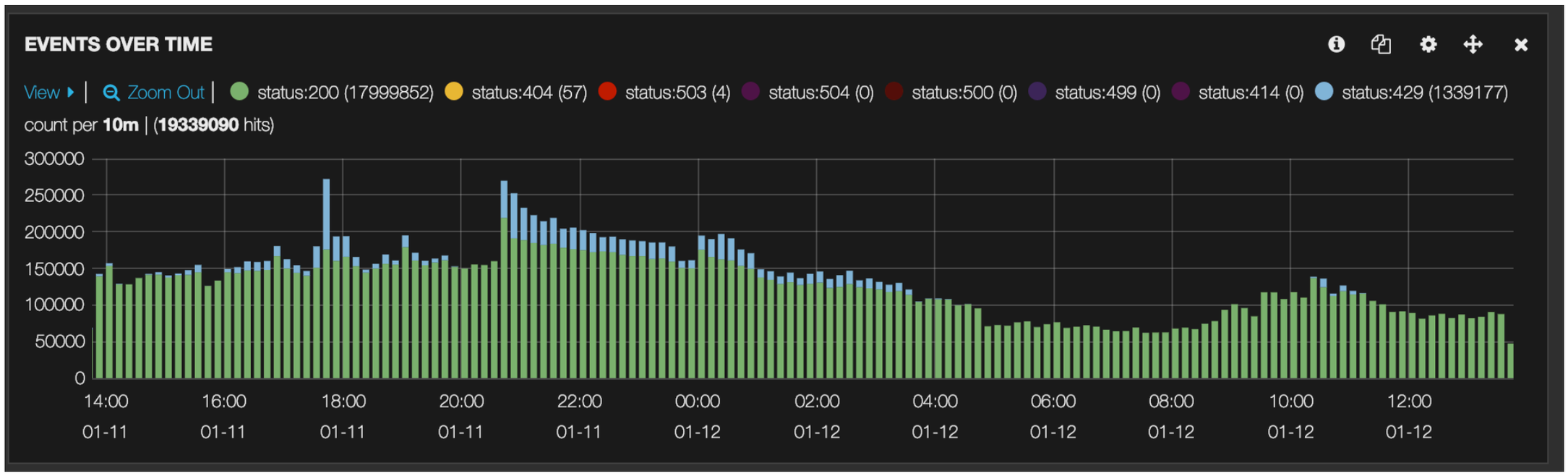
又拍云的线上实际案例 - 流量整形

```
body = function()
  local size = 8192
  if size > request_size then
    size = request_size
  end

  local count, err = token_bucket:take_available("service_id", size)
  if count == 0 then
    return nil, err -- drop body
  end
  size = count

  request_size = request_size - size

  local bytes, err = req_socket:receive(size)
  if err then
    return nil, err
  end

  return bytes
end
```



又拍云的线上实际案例 - 令牌桶应用

```
check_limit = {
    ["__global__"] = {
        interval = 100,
        capacity = 12000,
        quantum = 4,
    }, -- 40r/s 12000r/5m

    ["__bucket__"] = {
        interval = 200,
        capacity = 6000,
        quantum = 2,
    }, -- 10r/s 6000r/10m

    ["__delete__"] = {
        interval = 500,
        capacity = 1200,
        quantum = 1,
    }, -- 2r/s 1200r/10m
},
```



又拍云的线上实际案例 - IP 访问限制

限制规则

资源路径：/a/*.mp4

● 支持 * 作为通配符，如：/a/*.mp4 、/b/*.flv

限制方式：直接禁止

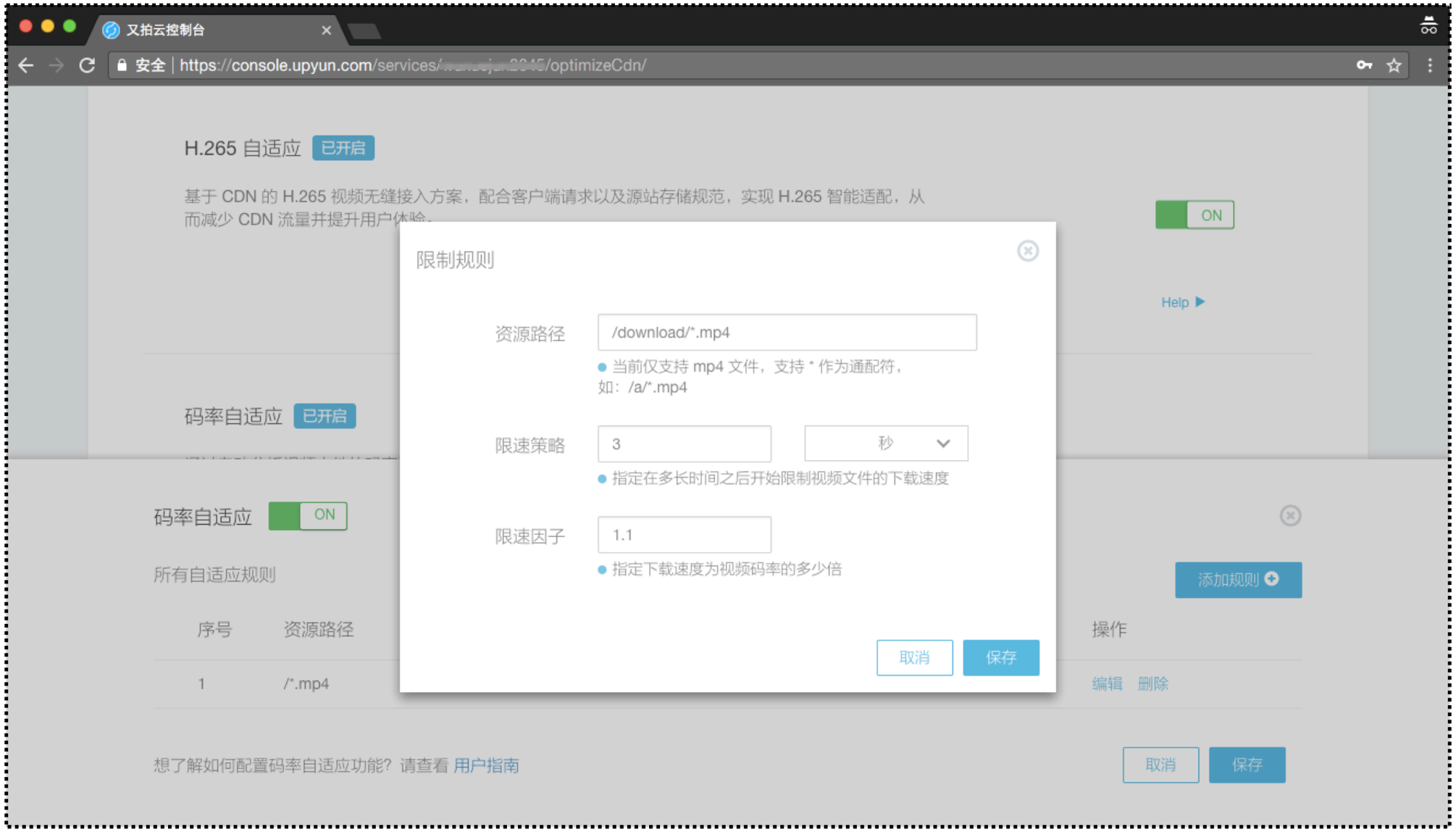
● 超出设定的访问频率，直接禁止 IP 访问

限制策略：

访问频率阈值（次/分）	禁止时间（秒）	操作
500 – 2000	60	⊕
2000 以上	300	⊗

取消　保存

又拍云的线上实际案例 - 码率适配限速



- 该功能通过自动分析当前视频文件传输的码率，将视频文件的传输速度控制在视频码率范围之内，可以在一定程度上防止高峰期时带宽占用，节省成本。

Q & A

又拍云