Charon: The Design of a Limiting Microservice

Eric Fritz Andy Brezinsky Andy Ortlieb
Mitel
{eric.fritz, andy.brezinsky, andy.ortlieb}@mitel.com

Abstract

Charon is a service designed to increase the stability of a distributed system by preventing the overcommitment of limited resources during extreme load. The service monitors the access history of resources and is used as a central authority which either *grants* or *rejects* requests for resource acquisition and use. This paper describes the architecture and feature set of Charon, as well as the rationale behind design decisions.

1. Introduction

A cloud service's resource utilization is a direct consequence of the number of active users and the types of activites they perform. The load placed on such resources can vary over time. For example, there is a likely increase in the number of active users during *Internet Rush Hour*, and expensive analytics computation or media transcoding may be performed in batch.

There may also be sudden and unpredictable bursts in activity, placing additional strain on resources. Such *bursty* behavior has historically been a scalability problem, as is evident from the following:

- Twitter reached a one-second peak of 143,199 Tweets/second on August 3, 2013 during an airing of the film *Castle in the Sky*. This consitutes a 2,400% increase over the average in per-second traffic [1].
- Michael Jackson's English-language Wikipedia page saw an unprecedented 5.9 million views in just 24 hours following his death in 2009. Prominent celebrity deaths dominate the top 100 traffic events for Wikipedia, according to Andrew G. West [2].

If resource utilization exceeds the system capacity, the service may suffer from increased latency, decreased throughput, or the failure of part of the service or the service as a whole. Clients and service providers participate in a Service Level Agreement (SLA), a contract outlining system-related characteristics such as expected service uptime and latency. For example, a service may guarantee that 99.9% requests receive a response within 200 milliseconds, given the peak client load does not exceed 500 requests per second. In the context of an SLA, such failures are unaccpetable and may have several consequences to revenue.

Autoscaling is a strategy to match provisioned resources with the current usage of a service at any given time by automatically increasing capacity to maintain performance during usage spikes, and decreasing capacity during usage lulls to reduce costs. While scaling of stateless tiers (e.g. webserver, load balancers, queue workers) is trivial, scaling of tiers holding persistent data (e.g. relational database) is more complex.

A database tier can be scaled horizontally by adding additional read slaves, but provides little benefit for write-heavy workloads. The provisioning of new a resource is not instantaneous, and in some circumstances may not be fast enough to react to demand. If the window of time between resource provisioning and resource availability experiences extreme load, some resources may fail under strain. Such failures tend to cascade and are likely to lead to system-wide downtime.

Throttling is an orthogonal strategy which allows the use of resources up to some *soft limit*, and then slow or reject subsequent requests for a period of time after this limit is reached. The service monitors resource utilization so that when an application and datacenter-dependent threshold is reached, subsequent requests from one or more users or subsystems are slowed or rejected. The remaining available resources can be partitioned appropriately, and a system under heavy load can continue to function (at a possibly *degraded*, but available, operational state).

Throttling can help to avoid overloading a particular subsystem either permanently or temporarily while additional resources can be provisioned. A common issue when dealing with distributed system failovers is the inability to recover a system because nodes are *continuously* failing-over under load. If a node fails due to high load, additional burden is placed on the remaining remaining nodes, create instability for an entire application tier. The following AWS service disruption post-mortems describe this situation in practice.

- A brief network disruption impacted a portion of DynamoDB servers. Normally, affected servers query for membership metadata after rediscovering the cluster before accpeting new requests. Due to recent orthogonal changes, the time required for this query neared then exceeded a retrieval timeout. Affected servers would remain offline, re-querying the metadata service. This placed additional strain on the service and began to affect additional DynamoDB servers that were not affected by the original network disruption [3].
- Amazon Elastic Block Store volume data is replicated across nodes for durability and availability. If one node cannot communicate with the node to which it replicates, it must choose another live node and re-mirror. Writes are blocked until replication completes. In two separate cases, a *re-mirroring storm* occurred after a large portion of EBS servers were disconnected from their replicas (due to a power failure [4] and a problematic network upgrade [5]). The remaining nodes could not find enough free space in which to replicate and were stuck blocking writes until additional capacity could be made available.

In the following, we introduce **Charon**. Charon is a centralized microservice which, according to a set of configured *access rules* and a user's *resource access history*, controls whether or not a user is allowed to use a particular resource. This service is *resource-agnostic*, and only monitors access to a resource by name, not by meaning. Clients that communicate with a shared resource (e.g. a database, a web API, a message queue, etc) register the use with Charon, which will either grant or reject access to the resource. A use may be immediate (e.g. database queries or web requests), or may be long-lasting (e.g. provisioning a sandbox to run user code).

Charon expects cooperative behavior as clients must voluntarily limit their own access. In a server-side limiting scheme, each request has already burnt network capacity and initial processing on a request before it determined that the request is over-limit. Client-side limiting prevents wasting time on such requests by first asking permission from a limiting server before sending the request to a shared resource, minimizing the cost of rejecting requests that are already over-limit.

This system provides a *central authority* of resource permissions. This enables resource limit configuration to be changed independently from code deployment and distribution, and allows distinct services to limit based on a single access history.

We have set two hard goals in the design and implementation of Charon, as follows.

1. The introduction of the Charon service to an infrastructure **must not** introduce a new single point of failure. The design of an appropriate Charon client **must** be able to make decisions autonomously when separated from the service (which composes the source of truth).
2. The response to a resource acquisition query **must** err on the side of *leniency*. The service **must not** approximate the behavior of clients in such a way that may lead to a request being incorrectly denied.

The remainder of this paper is organized as follows. Section 2 describes how limits are configured for resources and discusses common usage patterns. Section 3 presents the API of the limiting server, and discusses additional features in the reference client. Section 4 discusses architecture and implementation details and section 5 discusses related work.

2. Resources

Charon tracks resources which fall into two distinct categories. A resource can be tracked by *access frequency*, or can be tracked by *number of copies*. Access to the former resource is rate limited, and access to the later resource is guarded by acquiring a global counting semaphore. A rate-limited resource is granted according to uniform-cost *hits*, where the client may access to perform some operation after a hit is successful. A copy-limited resource can be *held* by a client for a variable amount of time. Such a resource is reserved when an access is successful, and released when the client has finished using the resource.

Access to a resource is requested by supplying the unique name of the resource as well as a globally identifiable *domain* name. The *domain* is most often a reference to the user or tenant for which the resource is requested, but could easily refer to other entities in a another context.

Each resource is identified by a globally unique name and are associated with a set of configuration parameters which control the rate and conditions of access to the resources. These parameters depend on the limiting technique used for the resource and are discussed in detail below. Resources also have a set of *domain-specific* configurations, which are given higher precedence than the *default* configuration - if a resource is requested for a domain and a domain-specific configuration exists, that configuration is used in the default configuration's place.

2.1 Rate-Limited Resources

Each rate-limited resource is configured with a *hard limit* H, a *global limit* G, and an ordered sequence of *burst tier configurations*. The *hard limit* and *global limit* of a resource describes the maximum number of requests that can be granted *per second* from a single domain and across all domains, respectively. If a hard limit or a global limit are not supplied for a resource, they are effectively unbounded.

A resource may have $n \geq 0$ burst tiers configurations (although when n is zero, the resource is inaccessible). Burst tiers are one-indexed ($i = 1, 2, 3, \ldots$). The ith burst tier configuration has the parameters

$$(L_i,\ T_i^w,\ T_i^a,\ T_i^c, S_i)$$

denoting the *limit*, the *window span*, the *active time*, the *cooldown time*, and whether the tier is *skippable*, respectively. A domain-specific configuration defines a distinct set of burst tier configurations, which may vary in number.

A *burst tier* describes a specific subset of a domain's access history for a resource, which is constrained according to the configuration parameters defined above. Two domains accessing the same resource will have distinct and non-overlapping access histories. Burst tiers and their behaviors are described in detail below.

Window At any point in time, the burst tier at index i maintains a *sliding window* which aligns on one end with the current time, and captures the last T_i^w seconds of activity. The hits which occurred beyond the bounds of this window can be forgotten, as they are irrelevant to any access decision.

The limit L_i denotes the maximum number of hits that can fall within the burst tier's window. If the current sliding window already contains L_i hits, no additional request can be fulfilled. As the sliding window updates implicitly with time, the next request can be granted only when

$$\texttt{now} > t + T_i^w$$

where t is the time of the *least* recent request within the sliding window.

State At any point in time, a burst tier is either *inactive*, in its *active period*, or in its *cooldown period*. The first hit within an inactive burst tier causes it to enter its active period. The burst tier at index i will automically transition from its active period to its cooldown period after T_i^a seconds have elapsed, and will in turn automatically transition back to its inactive period after an additional T_i^c seconds have elapsed. These transitions are illustrated in Figure 1.

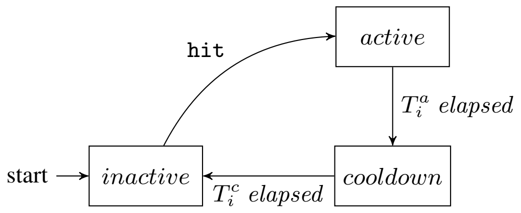

Figure 1. State transition diagram for burst tier i.

Moving Through Tiers The *current* burst tier is the active burst tier with the *greatest* index. There are circumstances which cause **no** burst tier to be in the active state (recent inactivity, a particular burst tier is in cooldown, or no burst tiers exist). In this case, we consider the *fake* burst tier at index zero to be the current burst tier. This fake burst tier has behaves as if it has the following configuration.

$$L_0 = 0,\ T_0^w = \infty,\ T_0^a = \infty,\ T_0^c = 0,\ S_0 = 0$$

The current burst tier may change with time through *bursting* or *fallback*. Fallback occurs when the current burst tier at index i transitions from its active period to its cooldown period. A lower burst tier at index $j < i$ which is still active automatically becomes the new current burst tier. Notice that fallback does not require that $j = i - 1$, and the current burst tier may drop several levels at once.

When a sliding window of the current burst tier at index i is at capacity, the next request may cause a burst into the next tier at index $i+1$. A burst cannot occur if there is no configured burst tier at index $i+1$, or if the burst tier at index $i+1$ is currently in its cooldown period and $S_{i+1}=0$. If $S_{i+1}=1$ and that tier is at capacity, then a burst may occur to the tier at index $i+2, i+3$, and so on, until a valid burst target is found. Bursting into this upper tier will immediately begin its active period. A request while in burst tier at index zero will always burst (when $n \neq 0$). Figure 2 illustrates an example of a successful burst and Figure 3 illustrates an example of a burst failure caused by a cooldown period.

2.2 Copy-Limited Resources

Each copy-limited resource is configured with a *domain limit* and a *global limit*. The *domain limit* of a resource describes the maximum number of copies a domain can hold at a given time, independent of other domains. A domain-specific configuration may redefine the domain limit for a particular domain. The *global limit* of a resource describes the maximum number of copies that can be held *across all domains* at a given time. If a global limit is not supplied for a resource, it is unbounded.

A resource may have $n \geq 0$ group-specific configurations, where each has a *domain group* (a set of domain names) and a *group limit*. The *group limit* describes the maximum number of copies that can be held by across all domains *within that domain group*. If a domain belongs to more than one domain group, then one copy of a resource is reserved for *each* group's pool of resources.

2.3 Cookbook and Patterns

We have found several useful resource configuration patterns, each discussed below.

Penalties Access patterns may be naturally *bursty* for many types of services. It is not necessarily a threat to infrastructure stability for bursts to occur, as long as the bursts do not become permanent. In order to accommodate short bursty access patterns, additional burst tiers can be stacked where the limit increases proportionally with the cooldown period. The higher a domain bursts for a single resource, the more they can consume - but only for a short period of time. After a period of indulgence, they are forced back down into a lower tier and cannot burst again until the cooldown period of the upper tiers have elapsed. This creates a small window in which a client can receive heavy traffic, but is forced to maintain a more even-paced pattern of access directly afterwards.

Punishment For other types of services, access patterns may be naturally uniform, and bursty behavior may indicate abuse or unstable clients. To mitigate bursty requests, a burst tier can be configured to *capture* bursty-behavior domains. A burst can be detected by either hitting a limit, or moving through an entire *buffer* tier (which may be entered, but only slightly, on normal access patterns). An upper *prison* tier can be configured with a limit of one and a long active period. Once entered, no further requests can be made by this domain for this resource, and the domain can only leave this tier after the active time elapses.

Batch Processing Most commonly, the lowest burst tier will have a zero-time cooldown period to prevent normal, non-abusive access patterns from suffering an iterating *hiccup*. However, breaking this convention can prove to be useful in certain circumstances. Suppose access to a hosted API is used for bulk processing, but isn't used frequently throughout the day. A single burst tier with the following parameters will allow up to 5000 API requests within a 5 minute window (300 seconds), but disallow any further access for the remaining 24 hours (86400 seconds).

$$L_1 = 5000,\ T_1^w = 300,\ T_1^a = 300,\ T_1^c = 86100,\ S_1 = 0$$

Bulkheads A ship's hull is divided into different watertight bulkheads so that if the hull is compromised, the failure is limited to that bulkhead as opposed to taking the entire ship down. Copy-limited resources can similarly be partitioned into several groups, where each group is meant to be acquired by a symmetric domain group. A problem (over-acquisition, in particular) in one group of resources will not negatively affect other groups.

A group of high-volume clients could acquire resources from a *reserved* set of copies, distinct from (or a subset of) the remaining copies used by lower-volume clients. One this set of copies is exhausted, high-volume clients are denied future requests, opposed to acquiring *all available* copies of a resource. This egalitarian approach prevents high-volume clients from drowning out the opportunities of smaller clients to acquire the same resource. Unknown clients can be placed initially into a *probationary* group with a non-critical number of copies shared with other new clients. Once they become trusted and their access patterns are better understood, they can be removed from the domain group and their limiting behavior changes immediately. A *blacklist* of domains can be given an arbitrarily low group limit, forbidding them access to particular (or all) resources.

3. API

This section details the request-response API via which the client communicates with the Charon service. Some additional features provided by the reference client are discussed in Section 3.2. These features demonstrate the utility of the API as low-level building blocks, allowing more complex limiting patterns to be built on top of the provided 'basic' functionality.

3.1 Server API

Clients communicate with the Charon service through a sequence of request-response transactions over a persistent gRPC connection. A client will receive exactly one response for each request issued (barring a network partition or a server crash). Each response is calculated *immediately*. That is, no request acts *asynchronously* and requires the client to ask or wait for supplemental data after the server's immediate response, and no response will cause the server to *retry* automatically on a rejection (or failure). It is left to the client to implement a retry mechanism in the event of a rejection.

Requests are, for the most part, designed to be stateless. A client must supply as much context as necessary in order for the server to understand the request in-full. However, for reasons elaborated in Section 4.3.1, each server acts as the arbiter for a client's hold on a copy-limited resource and must maintain some in-memory bookkeeping with respect to each client. As a consequence, client sessions are *sticky*, and a client is bound to a single server for (at a minimum) the lifetime of the resource they hold. Clients may issue many independent requests using the same session, and are encouraged to do so to minimize transport overhead.

In addition to the set of normal responses for each request, the server may respond with one of two classes of *error* responses. A *server error* response indicates an internal problem, most likely an issue connecting to a dependent service. A client should not immediately re-issue the request to the same server. A *client error* response indicates a problem with the request itself. This can happen for a variety of reasons, including:

1. The request message is not well-formed. This is a class of syntax error which occurs when the server cannot extract the necessary information from the request packet.

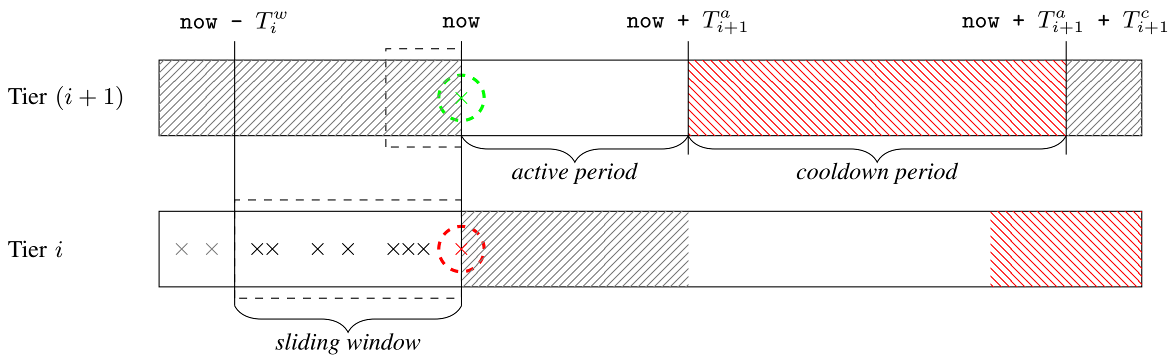

Figure 2. Successful bursting.

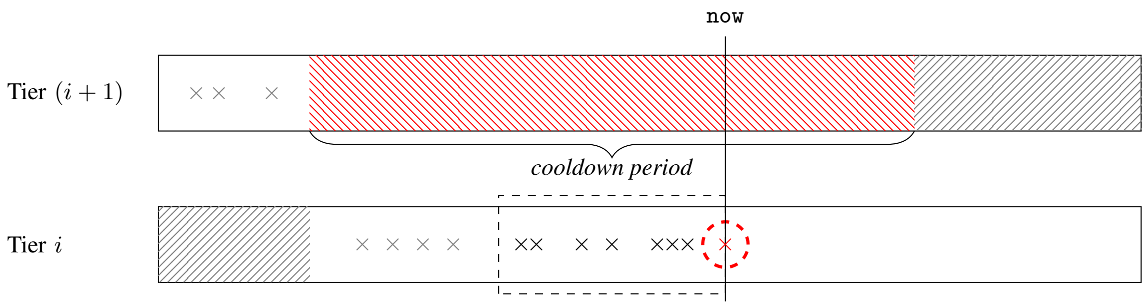

Figure 3. Burst blocked by cooldown.

2. The request refers to a resource, but the server is not aware of any configuration for that resource.
3. The request refers to a resource, but the server is aware of configuration which contradicts the request itself (requesting a copy-limit a rate-limited resource, and contrariwise).
4. The request attempts to release a resource, but the client issuing the request does not currently hold that resource.

The server API includes the following five low-level operations, each acting with respect to resource r on behalf of domain d. Each operation is designed with simplicity and minimal latency as top priorities. The `request` operation asks for a rate-limited resource. The `reserve` operation followed by an eventual symmetric `release` operation denotes the hold of a copy-limited resource. The `transfer` and `seize` operations allow transfer the resource holds to another client.

request(`r, d, copies, min_copies`)
The `request` operation is the only operation which applies to rate-limited resources. The server will attempt to apply n hits to the access log for resource r from domain d, where n is the maximum allowable value in the range $[min\_copies, copies]$ respecting the configured hard limit, global limit, and the state of the current burst tier for this domain and resource pair. If only $n < min\_copies$ can be allocated for this resource, the request will fail and the access log will remain untouched.

As multiple hits can be logged at once, it is possible that a single request may pass *completely through* an entire burst tier (or multiple tiers, for very large bulk requests). If min_copies exceeds the hard limit of the resource, the request will always fail.

In general, $min\_copies = copies = 1$, but it may be appropriate to bulk-request a number of resource hits up-front and abort if the minimum number of requests cannot be made. This operation acts atomically and cannot be emulated by multiple requests as rate-limited access cannot be *un-requested* after a grant.

The server response payload contains n, the number of hits to the resource granted to the domain, and the following context data. On rejection, $n = 0$.

- The resource's hard limit
- The resource's global limit
- The domain's active burst tier limit
- The number of hits within the last second by this domain
- The number of hits within the last second across all domains
- The number of hits in the domain's active tier
- The index of the domain's active tier
- Whether or not the request caused a burst
- Whether or not the request was limited by a hard limit
- Whether or not the request was limited by a global limit

reserve(`r, d, copies, min_copies`)
The `reserve` operation (as well as all the following operations) applies to copy-limited resources. Similarly to `request`, the server will attempt to hold n copies of resource r from domain d, where n is the maximum allowable value in the range $[min\_copies, copies]$ respecting the configured global, domain group, and domain limits. If only $n < min\_copies$ can be reserved, the request will fail.

In general, $min\_copies = copies = 1$, but it may be necessary to act on several copies of a resource at once. This operation adds atomic semantics to multiple calls to `reserve`. This operation can be emulated with multiple requests, but unnecessary resource access would be held for a period of time in the case of (eventual) rejection.

The server response payload contains n, the number of copies of the resource granted to the domain, and the following context data. On rejection, $n = 0$.

- The resource's domain limit
- The resource's global limit
- The number of holds by this domain
- The number of holds across all domains
- For each domain group to which the domain currently belongs:
  - The group's name
  - The group's limit
  - The nubmer of holds across all domains in the group

The name of the groups will be necessary for subsequent `release` and `transfer` operations, discussed below.

release(`r, d, copies, groups`)
Each successful reservation of a copy-limited resource should be paired with a symmetric `release` operation following its use. The request message contains the number of copies to be released as well as the set of domain groups to which the domain belonged *at the time these copies of the resource were reserved*. A single resource and domain pair can affect the number of holds differing sets domain groups, as a domain can be added or removed from a domain group while it is actively holding resources. The domain group must be supplied to the server in order to disambiguate which set of resources should be released.

A `release` operation **must** be issued to the same server to which the symmetric `reserve` operation was issued. It is an error to attempt to release more copies than are reserved by the given resource, domain, and domain group.

If a reservation does not have a symmetric release, then the reserved copies of the resource are *leaked* and cannot be reserved by another client. We show how the effect of this issue is mitigated in Section 3.2 (due to neglect) and in Section 4.3.1 (due to exceptional conditions).

While all copies of a resource must be *eventually* released, they do not need to be released in bulk. A set of resources reserved in a single request can be released over a number of requests, as long as the the copies being released sum to the number of copies reserved.

The server will reply with an empty acknowledgement on success. There is no normal failure condition for this request - any error is a client error, or a fault of the server.

transfer(`r, d, copies, groups, ttl`)
A `transfer` operation is the first step in moving a subset of holds from one client to another. A `seize` operation, discussed below, is the second and final step in the transfer.

A transfer is useful if the use of a resource spans multiple processes or machines, and an accurate tracking of the number of resources in-use is required. This operation does **not** affect the domain or group for which the hold applies - it simply allows the hold to re-bound to another TCP connection, possibly to a TCP connection on a separate instance of the Charon service.

A `transfer` operation is similar to a `release` operation, but instead of the copies of the resource being moved back into a free pool, they are moved into a *transfer staging area*. A unique $transfer\_id$ can identify this set of resources. If the resources are not moved out of this staging area, they will be released implicitly after ttl seconds have elapsed. Resources staged for transfer are still implicitly held and cannot be reserved by any client, but can no longer be explicitly released by the client.

The server response payload includes the unique $transfer\_id$ which identifies the set of resources in the staging area. As with the `release` operation, there are no normal failure conditions for this request.

seize(`transfer_id`)
A `seize` operation allows a client to claim a set of resources from a transfer staging area, identifiable by the given $transfer\_id$. The server response payload will includes the following context data.

- The resource namespace
- The resource name
- The domain name
- The list of groups to which the domain currently belongs
- The number of copies seized

This context data is enough to reconstruct a client-side hold on a resource *indistinguishable* from resources which were acquired through a `reserve` command. Seized resources must be released or transferred by the client at the end of its use.

will reply with a `seized` response which includes the name of the resource, the number of copies, and the new domain which holds the transferred resources, as well as the set of domain groups to which the domain currently belongs.

It is an error to attempt to seize a $transfer\_id$ which does not exist or has expired (which happens when its ttl has already elapsed). These error conditions may not be distinguishable in practice.

3.2 Client API

The reference client, written for Python 2.7, is a thin abstraction over the server API presented in Section 3.1 with the notable exception of two usability features which are discussed in this section.

The first feature is a mechanism which re-issues `request` and `reserve` operations in the face of rejection. This mechanism obeys a maximum wait time, max_wait, which is supplied by the user. By default, the maximum wait time has a value of zero, and requests are not re-issued. The request is issued in a loop according to an *exponential backoff policy*. After the ith failure, the client will issue a sleep for

$$\min(max\_wait - elapsed,\ jitter(2^i))$$

seconds before attempting the request again, where $elapsed$ is the true time spent blocking, and $jitter$ is a function which randomizes a scalar value v uniformly within $[v - \frac{v}{4}, v + \frac{v}{4})$. Once the sleep target dips below zero, the request is aborted and the last failure issued is given back to the user. This policy, opposed to immediate retries, heavily reduces the number of futile requests which are unlikely to receive a grant, which become increasingly common as load increases. From the user perspective, the call is blocking at the server level. Use of a maximum waiting time is illustrated in Figure 4.

```
# (blocks for at most 5 seconds)
res = client.request_rate(r, d, max_wait=5)

if res.success:
    handle_api_call()
else:
    raise OverLimitError()
```

Figure 4. Retry with maximum wait time.

The second feature is an *implicit release* of a resource after executing the block of code which requires it. This feature uses Python's *context managers*, which will invoke a `release` command for the remaining resources which have not been released explicitly by the user. The use of a copy-limited resource is illustrated in Figure 5.

This ensures that the user does not forget an explicit release, and does not need to concern themselves with guarding the block of code to capture exceptional exit conditions. However, a client which faults (ending the process exceptionally) during this code block may never perform the release. The circumstance of a faulting client is discussed in further detail in Section 4.3.1.

```python
with client.hold_copy(r, d, copies=10) as res:
    if res.success:
        step1()         # has 10 copies
        res.release(5) # release 5 copies
        step2()         # has 5 copies

# (remaining copies implicitly released here)
```

Figure 5. Implicitly releasing holds of resource.

Figure 6 illustrates a multi-process exchange of resource copies. This example assumes the existence of a socket or another communication mechanism which can send and receive the transfer identifier from one process to another.

3.3 Client-Side Cookbook and Patterns

We have found several ways to compose server operations to make expressive higher-level operations which have proved useful in practice, each discussed below.

Multi-Resource Requests Although the server API only allows for multiple *copies* of a resource to be requested atomically, we can combine several requests on the client-side to *pseudo-atomically* request copies of *multiple* resources (providing client-observable *atomicity*). This pattern works well when requesting m resources where *at most* one of the resources are rate-limited. Suppose at least min and at most max copies of the resources $R = \{r_1, \dots, r_m\}$ must be reserved. The *critical section* can be entered only when **exactly** the same number of copies are reserved for each resource r_i. Additionally, suppose that the request for these resources must complete within max_wait seconds.

First, request max copies of r_1 with a max wait of max_wait. If successful, the response will contain n_1, the number of copies of r_1 which are now held (which is necessarily greater than min). Next, request n_1 copies of r_2 with a max wait of $max\_wait - elapsed$, where $elapsed$ is the time spent on the first request. If successful, the response will contain n_2, the number of copies of r_2 which are now held. Now, $min \leq n_2 \leq n_1$, but it may be the case that $n_2 \neq n_1$. In this case, we release $n_1 - n_2$ copies of r_1 before continuing. We continue this pattern through r_m, releasing extra held copies along the way. If R contains a resource which is rate-limited, it must be requested as the final step. Otherwise, the resource has been over-allocated and cannot be released back to the server.

Figure 7 demonstrates this pattern with a copy-limited resource $\texttt{res}_c$ and a rate-limited resource $\texttt{res}_r$.

4. Architecture and Implementation

The Charon service is split into two subsystems: the configuration API and the limiting server. These subsystems are decoupled, which allows them to scale independently. Each subsystem communicates through its public interface through a proxy. This allows horizontal scaling to remain opaque to the user.

4.1 Configuration API

The configuration API is a simple HTTP API backed by a relational database. This subsystem acts as the central authority for resource configurations, which were discussed in Section 2.

The API allows the creation and mutation of resource configurations, but happens to *normalize* the data before it is committed. Normalization of resource configuration includes (but is not limited to the following changes.

- Burst tiers with an active time of zero are removed.
- If the active period of a burst tier is smaller than that burst tier's window, the window can be shortened to the size of the active period.
- If the active period is not a multiple of the burst tier's window span, the active period can be trimmed so that no *partial windows* occur at the tail of the active period. If partial windows are allowed at the tail of a burst tier with a cooldown of zero, this presents a possibility for a user to go over-limit when crossing into the second instance of the burst tier.
- If the limit of a copy-limited resource, including domain and group overrides, exceeds the global limit, the *effective limit* is lowered for that resource or override.

The configuration API is required by the limiting server, and is accessible externally for clients which create and update the configuration for infrastructure resources. Although the limiting server communicates with the configuration API, it does not do so frequently. Each limiting server maintains its own *view* of the resource configurations in-memory and will periodically (or on-demand) read the configuration API for changes in order to refresh their stale view. The configuration API can send a delta with respect to the limiting server's current configuration view in order to minimize the cost of updating internal structures.

Mutation of resource configurations are expected to happen rather infrequently in practice. New resource configurations are added at the speed of code deploys. Old resource configurations are updated after discovering that a limit was set too high and users were utilizing resources too frequently or holding too many copies concurrently, or a limit was set too high and the Charon service was responding with spurious rejections.

4.2 Limiting Server

The limiting server is a high-concurrency, low-latency TCP server which implements the command interface discussed in Section 3.1. When a client connected to the limiting server, a *client session* is created. This session is bound to a particular limiting server for the life of the client. Resource holds cannot outlive the session from which they were created, unless they are explicitly transfered to another client session.

Each limiting server has a set of soft dependencies on services used for monitoring. Logstash is used for log aggregation between multiple instances of limiting servers. Riemann is a high-performance network event stream processor which is used for monitoring. The details of each request (the target resource, domain, and the result of the request) are sent to Riemann for monitoring. This data can be processed and analyzed in order to find access pattern trends and to adjust resource configurations as necessary.

Each limiting server has a hard dependency on Redis [6], an in-memory data structure store, which is used as the *shared memory* between limiting servers within a datacenter. Redis is implemented as a key-value store, where keys are unique strings and values are one of several data structures (e.g. string, number, set, sorted set, hash). The data stored in Redis is *authoratative*, and limiting

```
# Process #1
client = Charon(port=54648)

with client.hold_copy(r, d, copies=15) as res:
    if res.success:
        step1()                             # has 15 copies
        tid = res.transfer(10, ttl=30) # stage 10 copies
        send(tid)                           # (to Process #2)
        step2()                             # has 5 copies
```

```
# Process #2
client = Charon(port=54649)

tid = recv() # (from Process #1)
with client.seize_copy(tid) as res:
    if res.success:
        step3()          # has 10 copies
        res.release(5) # release 5 copies
        step4()          # has 5 copies
```

Figure 6. Transfer of resources from one client to another client in a different process, each connected to a difference instance of the Charon service.

```
with client.hold_copy(r_c, d, copies=n, max_wait=max_wait) as res_c:
    if res_c.success:
        n = res_c.copies                        # Only request as many copies as we've been given
        max_wait = max_wait - res_c.waited # Only count time we already waited once in total
        res_r = client.request_rate(r_r, d, copies=n, max_wait=max_wait)
        if res_r.success:
            extra = res_c.copies - res_r.copies
            if extra > 0:
                res_c.release(extra)        # release extra copies
            process(res_r.copies)           # process remaining copies
```

Figure 7. Requesting multiple types of resources *pseudo-atomically*.

servers query the store on each request to determine the current state of the access and copy logs. Limiting servers do not act on their own *view* or a snapshot of the access or copy logs, which could quickly become stale.

Each request to the limiting server may perform several low-level reads and writes to the access and copy log data structures within Redis. Mutation of a data structure should only occur in the case the limiting server responds with a success to the client. This series of reads and writes are implemented as a *module* – a C-library loaded dynamically into the Redis process. As Redis's non-networking code runs in a single thread, the module can service an entire request atomically.

Other rate limiting systems using Redis as a data store must be careful to order sequences of get and set operations such that the classic race condition (read in process A, read in process B, write in process A, write in process B) does not occur. This either requires acquiring a lock around relevant keys (incurring additional lock and unlock operations on each request) or ensure that all counter operations are incremented prior to the deciding read (which increments a counter even on failed requests) [7, 8].

Access Log The *access log* is structured as a snapshot of the burst tiers at the point of last access with respect to a resource and domain. Each burst tier is represented by the timestamp of the most recent *entry time* into the tier, as well as a *set of successful hits* ordered by time. The entry time, current time, and the burst tier configuration are enough to determine the current *state* (e.g. active, inactive, cooldown) of the burst tier. Keys associated with a burst tier whose cooldown period has already ended are removed from the access log. On each access of a burst tier, elements falling outside the current window are removed from the ordered set which allows the size of the window to be implemented as a simple set-cardinality operation.

Pruning hits from an ordered set takes time proportional to

$$\mathcal{O}(log_2(k) + r)$$

where k is the number of hits in the set and r is the number of hits falling outside the current window. Notice that $r \leq k$, and k is bounded by the limit of the window. This operation is very efficient in practice as r remains small with frequent pruning.

Querying the access log takes time proportional to

$$\mathcal{O}(\sum_{i=1}^{n} log_2(L_i) + (log_2(G) + G) + \max_{i=1}^{n}\{log_2(L_i) + L_i\})$$

where n is the number of burst tiers and G is the global limit of the resource. The terms above come from, in order, *(1)* counting the number of hits within the last second of each burst tier, *(2)* pruning the global hit set, and *(3)* pruning the hit set of the domain's active tier. The linear term G amortizes with frequent requests for the resource, and the linear term L_i amortizes with frequent requests for the resource from the same domain.

Copy Log The *copy log* is much simpler. Each resource has a count associated with its global usage, its usage per domain group, and its usage per domain. Each hold of a resource increases one or more of these values (depending if a global limit is configured or if the domain belongs to one or more domain groups), and a release symmetrically decreases one or more of these counts.

Requesting, seizing, and releasing a resource takes time proportional to $\mathcal{O}(g)$ where g is the number of domain groups to which the domain belongs. A transfer/seize pair is a constant-time operation, but an unsuccessful transfer acts identically to a release.

Limiting servers associate a *hold set* with each client session. A hold set is a list of holds (number of resources) which were created from this session indexed by resource, domain, and domain group. Reserving a resource increments a count in the hold set, and releasing a resource decrements a count in the hold set. This information is *redundant* and is used to clean up after clients which have disconnected from their session while they are holding resources, as discussed in Section 4.3.1.

Scaling If limiting servers is scaled horizontally, the resulting *cluster* of limiting servers has an additional hard dependency on ZooKeeper [9, 10] to coordinate the discovery of other nodes in the cluster. While limiting servers do not need to communicate directly, they do need to be alerted when a limiting server goes offline (from either a network partition or a server crash). This scenario is discussed in detail in Section 4.3.2.

4.3 Failures

The limiting server, and in particular the data model, has been designed around the fact that every service, every dependency, and every edge in the network graph will at some point inevitably fail in a spectacular way and at an unexpected time [11, 12]. A particular concern which arises in the face of failure applies to the data consistency of the copy log. Particular failure circumstances and their resolution are discussed in Section 4.3.1 and Section 4.3.2 below.

A group of resources (distinguished by a namespace prefix) can be configured to *ignore* configured resources and to *allow unconfigured* resource access. These settings are orthogonal and can be enabled or disabled independently. Ignoring configured resources effectively disables limiting for a particular application. This can be useful in practice if there is a limiting misconfiguration which is not immediately resolvable (which is possible if the misconfiguration is too large spanning multiple resoruces or domain overrides, or if the misconfiguration is not immediately obvious).

If unconfigured resource access is enabled, then unknown resource configurations will behave as if they are configured with the *most permissive* values possible (e.g. maximum value limits, no cooldown, short window periods, and no global or hard limits). Resource accesses are still recorded. When an application introduces a limit to a resource which was previously unlimited, it may be difficult to configure limits for the resource in a way that is neither too restrictive nor too permissive. This feature can be used to log production access data for a period of time to determine a reasonable limit.

The clients are also designed with two *degraded* modes of operation. If the client cannot reach a limiting server, or if the limiting server returns an (non-client) exceptional response, the client will fail-open and begin operating in degraded mode. When this happens, the client will forcibly grant the request, but for the minimum amount of copies or hits that will allow the process to continue successfully. **It is imperative that a client application can remain operational, even if the limiting servers are completely removed from the network topology**.

The client also has a global *kill switch* that, when activated, will transform rejections from the limiting server to grants. This mode is used internally to strange behavior in limiting servers with live traffic and production limit configuration, while allowing the client to behave as if the limiting servers were behaving correctly.

4.3.1 Client Fault

A client may fail exceptionally without releasing its active holds. This can happen if the client process or the thread actively holding the resource encounters an exception and does not run the block finalizer, or can happen if the client simply loses connection to the limiting server.

If no action is taken in this circumstance, then the resources held by that client are held permenantly by a client session which no longer exists. These resources are effectively *leaked* and cannot be re-reserved in the future by any other client.

When a limiting server loses connection to a client, either from an explicit disconnect or a broken pipe, the client session is terminated. The client's hold set contains an in-memory list of all the outstanding holds the client has yet to release. Upon termination, the hold set is iterated, and each active hold is released as if it were released explicitly by the client. The server performs one release operation for each unique resource, domain, and domain group hold (not for each held copy of the resource).

4.3.2 Limiting Server Faults

A limiting server may fail exceptionally, most likely due to a losing access to one of its critical dependencies. When a limiting server fails, each connected client is no longer attached to the limiting *cluster*. This has the same effects as the client itself detaching from the limiting server.

To prevent the leaking of resources held by *every* client of a failed limiting server, the resources in the hold set of every connected client must be released. Unfortunately, the hold set information is no longer accessible, as it was stored in the memory of a server that is no longer running.

In order for this information to persist after a crash, each server maintains a redundant set of keys prefixed by the server's unique identifier in the copy log that specifies their *contribution* towards a global count. A count and the server's contribution are incremented and decremented in unison, and any count in the copy log should be the sum of each server's contribution towards that count. Each server also stores a set of *raw keys* to which they have contributed a non-zero count.

As a concrete example, suppose server s_1 has connecte clients c_1 and c_2 representing domains d_1 and d_2, respectively, and server s_2 has a connected client c_3 also representing domain d_2. If c_1 holds 4 copies of r, c_2 holds 5 copies of r, and c_3 holds 6 copies of r, then the copy log would appear approximately as the following. The same technique is performed for the domain group and global

$$\begin{aligned} r : d_1 &\rightarrow 4 \\ r : d_2 &\rightarrow 11 \\ s_1 : r : d_1 &\rightarrow 4 \\ s_1 : r : d_2 &\rightarrow 5 \\ s_2 : r : d_2 &\rightarrow 6 \\ s_1 &\rightarrow \{r : d_1,\ r : d_2\} \\ s_2 &\rightarrow \{r : d_2\} \end{aligned}$$

counts for the same resource.

When a limiting server crashes, the *leader* of the limiting server cluster is responsible for clean-up. The leader can find the server's set of raw-keys in the copy log. Each raw key is decremented by the server's contribution to that key. Then, all keys prefixed with the server identifier can be deleted.

To continue the example above, if s_1 crashes, then $r : d_1$ is decremented by the count stored in $s_1 : r : d_1$, and $r : d_2$ is decremented by the count stored in $s_1 : r : d_2$. Then, any key starting with s_1 is removed. This retains the 6 holds of r by d_2, and the copy log would appear approximately as the following. In practice, keys with a zero count are pruned upon decrement (hence the missing key $r : d_1$).

$$\begin{aligned} r : d_2 &\rightarrow 6 \\ s_2 : r : d_2 &\rightarrow 6 \\ s_2 &\rightarrow \{r : d_2\} \end{aligned}$$

If the *last* limiting server of a cluster crashes, then there is no leader which can be elected to clean up the copy log. This case is problematic when additional limiting servers subsequently connect to the same copy log containing now incorrect hold data. If a limiting server is the *only* server connected to the cluster on startup, the server begins by removing every key from the copy log so that no possibly stale data persists from a previous generation of limiting servers.

If a limiting server loses its heartbeat connection to ZooKeeper for a long enough period that the ZooKeeper session expires, the other limiting servers in the cluster will be informed that the server is no longer reachable. In this case, it would be incorrect for the server to remain operational, as its portion of the copy log has been forcibly removed. The in-memory hold set and the copy log are no longer properly synchronized, and subsequent release operations can cause the number of held resources to go negative. If a limiting server's ZooKeeper session expires, then the server *resigns*, shutting itself down immediately to prevent the copy log from being damaged. If limiting servers scheduled by a cluster manager such as Mesos or Kubernetes, then this case is generally observable as a small, self-resolving hiccup.

5. Related Work

This section discusses a few similar systems which are either in-production or available as open source.

5.1 Doorman

Doorman, open-sourced by members of the YouTube team in 2016, is a similar co-operative rate limiting system that achieves eventual stability despite a changing number of clients and dynamic load and capacity [13]. In this system, clients contact a limiting server to get a *lease* on the capacity of resource until a particular time T, after which the lease expires. It is the client's responsibility to conform to the granted lease, and the client must renew its lease with the limiting server before the current lease expires to continue using the resource. The capacity and limits for a resource, which are dependent on dynamic factors (e.g. the number of active clients, current load, etc), are re-evaluated regularly by the system.

A limiting server partitions and distributes the total capacity of a resource among clients while attempting to conform to the following properties.

1. *(No Over-Commitment)* The combined capacity given to each client does not exceed the total capacity of the resource.
2. *(No Under-Commitment)* All available requested capacity is distributed - the limiting server does not leave capacity on the table if there is a request for it.
3. *(Equal Distribution)* The available capacity is distributed *fairly* between clients, dependent on a pluggable algorithm which defines *fair*.

Each resource can be configured with a *safe capacity* which dictates the behavior of clients when Doorman cannot be reached for a lease. If this capacity is -1, then rate limiting is disabled. If this capacity is 0, then the use of the resource is forbidden. Otherwise, the safe capacity is used as if it was the capacity returned by a limiting server. While Doorman is active, a safe capacity can be calculated dynamically and given to clients with each lease.

Distribution algorithms use the configured capacity of the resource, the client's outstanding lease, and the requested capacity of the clients to partition capacity among clients. In addition to conforming to the properties listed above, an algorithm never gives capacity to a client exceeding their requested capacity. While other algorithms are possible, the five following algorithms are provided.

None Gives each client its wanted share, does not limit. This 'pseudo-algorithm' acts as a no-op distribution algorithm and may overcommit.

Learn Gives each client the capacity it claimed it had before. This 'pseudo-algorithm' is used when a new limiting server is being brought online and does not have a consistent view of the existing leases.

Static Gives each client a statically assigned capacity. This algorithm overloads the definition of *capacity* to mean a per-client maximum capacity rather than a global maximum capacity.

ProportionalShare Every client is given up to an equal share of the available capacity. Any capacity left on the table by clients requesting less than their equal share is distributed among the remaining clients proportionally to their requested capacity. For example, a resource with a capacity of 90 will distribute in the following way. Each client gets one-third of the capacity, but Client C leaves 20 shares on the table. These shares are divided among the remianing clients such that Client A gets 78% ($\frac{70}{90}$) and Client B gets 22% ($\frac{20}{90}$) of the remaining shares.

Client	Requested	Share	Additional
A	100	30	15.6
B	50	30	4.4
C	10	10	0

This algorithm takes time proportional to the number of clients.

FairShare Every client is given up to an equal share of the available capacity. Any capacity left on the table by clients requesting less than their equal share is distributed equally among the remaining clients in rounds so that as many clients as possible get their requested capacity. For example, a resource with a capacity of 160 will distribute in the following way. Each client gets one-fourth of the capacity, but Client D leaves 30 shares on the table and the process is repeated for clients with outstanding requested capacity. Clients A, B, and C each get one-third of the remaining shares. This fulfills the requested capacity for everyone except Client A, who gets the leftover shares.

Client	Requested	R1	R2	R3
A	100	40	10	5
B	50	40	10	0
C	45	40	5	0
D	10	10	0	0

This algorithm takes time quadratic to the number of clients and may not be suitable for a resource with a large number of distinct clients.

Doorman moves the responsibility of granting or rejecting individual operations to the client, as the limiting server monitors only granted client capacities, not use of that capacity. This requires that clients know or predict the capacity they will require for the extent of a lease, which may be inaccurate. Clients must be fitted with adaptive logic so that they are neither over-requesting or under-requesting capacities, otherwise clients may waste capacity that could be used by another client, or clients may be artificailly limited.

Doorman likely produces less network chatter, as services only need to communicate when a lease expires. Charon, on the other hand, requires that each access of a resource is preceded by a decision from a limiting server. As an optimization in this direction, Charon allows multiple resources to be granted at one time (as discussed in Section 3.1).

Clients in Doorman and Charon also ask for resource access within fundamentally different roles. When Doorman grants a lease to a client, it does so specifically for the service. When Charon grants resource access to a client, it does so *for a specific domain*. As a consequence, Charon is applicable for PaaS architectures where resource usage from individual *tenants* or *end-users* are logically distinct, where Doorman is not.

5.2 Ratelimit (Lyft)

Lyft uses a similar gRPC service written in Go for rate limiting [14]. Clients request access to a resource for a domian and a set of *descriptors*, which are a list of hierarchical entires used to determine a final key held either in-memory or a Redis cache.

```
domain: messaging
descriptors:
  - key: to_number
    rate_limit:
      unit: day
      requests_per_unit: 100

  - key: message_type
    value: marketing
    descriptors:
      - key: to_number
        rate_limit:
          unit: day
          requests_per_unit: 5
```

In the example configuration shown above, a domain *messaging* is defined with two top-level descriptors. The first descriptor with key *to_number* has no value, so each unique value suppied for this key will be applied under a unique limit. Thus, $(to\_number, 4145551234)$ and $(to\_number, 4145551235)$ may *each* generate 100 requests per day. The second descriptor with key *message_type* has a concrete value *marketing* which matches only requests with the same concrete value. Thus, a request with the descriptor list

$$(message\_type, marketing), (to\_number, 4145551234)$$

may only generate 5 requests per day (limited orthogonally to the other descriptor lists).

This system of configuration appears to be quite powerful and seems to generalize Charon's ability to apply limits to a single domain, to a group of domains, or globally. Extending Charon's configuration so that arbitrary request data may match a different limit configuration is possible, but is not yet implemented.

Lyft integrates this system into their distributed service mesh *Envoy* in two locations. First, the network level filter calls the ratelimit service for every new connection on the proxy's socket listener. This allows limiting of the number of connections that transit the socket listener. Second, the proxy calls the ratelimit service for every new request on the listener. This allows limiting on the application level where additional client data is known by the proxy.

5.3 Cloud Bouncer

Yahoo integrated Cloud Bouncer, a largely decentralized rate limiting solution, into their platform in 2014 [15]. The software can be decomposed into two parts. The *policy manager* is a database-backed API where users can register their applications and rate limiting policies. Policies can be based on any attribute of a request, including the distinction between authenticated or anonymous requests, read or write requests, and the number of bytes uploaded. The *controller* is a backgroud process which is ran on each node where rate limiting is enabled. The controller periodically polls the policy manager and caches the current policy state in memory – the same process Charon has for limiting configurations. The controller process makes a decision locally of whether or not to serve the request without any external communication during the request. Traffic information for each node is stored locally and communicated to every other node in the cluster via the gossip protocol over UDP. In order to reduce network chatter, controllers are optimized to send only the last second of traffic data to the rest of the cluster.

5.4 Kong

Kong is a microservice API gateway and service mesh that provides rate limiting of downstream (non-edge) APIs [8]. In order to reduce the latency of each limited request, each gateway node makes a local determination in-memory. Periodically, each node push its actions since the last synchronization to a shared data store and read the updated values. This has the effect of relaxing the limiting condition so that a client can exceed the limit between node synchronization cycles. Individual nodes may be inaccurate for a period of time, but will eventually reconverge to a nearly-accurate global view.

5.5 PoolCounter

MediaWiki uses a network daemon which provides mutex-like functionality called PoolCounter [16], which was created to lessen CPU wastage when may servers are required to compute the same result in parallel. This class of problem is common in Wikipedia's case (coined the 'Michael Jackson Problem' [2]), where articles with a sudden spike of popularity and frequent edits cause a large number of servers to re-parse the same page in order to serve fresh content.

A client opens a connection to PoolCounter, sends a lock acquire command, does the work, send a lock release command, and disconnects. When a lock is released, a process waiting in the wait queue may be woken with an acquired lock. PoolCounter is equipped with a limited wait queue length and a client that overflows the wait queue must perform a request-specific fallback command (e.g. serve a stale cache entry or display an error message). Two notable lock acquire commnads are defined.

ACQ4ANY This is used to acquire a lock when the result of this computation is transferrable across processes via a shared cache. When a lock of this type is released, all waiting processes are awoken (without an acquired lock) to read the new cache entry.

ACQ4ME This is used to acquire a lock when sharing of a cache entry is not applicable. When a lock of this type is released, only one waiting process is awoken with an acquired lock in order to keep the worker population at a constant maximum.

5.6 Smockron

Smockron is a distributed rate limiting service that inverts the request/grant-or-reject framework discussed so far [17]. Smockron is a leader-follower cluster which communicates to application servers via ZeroMQ. The application server sends a metadata payload to a limiting server for each incoming request, but does not wait for a grant/reject acknowledgement from the limiting service. Instead, when the limiting server detects (asynchronously) that a client is using a resource too frequently, it sends a message to each application server instructing them to deny the client for a period of time.

This inversion of control is an interesting alternative with a number of benefits. Most beneficial seems to be that the limiting request/response cycle imposes no additional latency on client requests as an initial determination can be made in-process and communication with the limiting server can be done out-of-band. Additionally, the fail-open mindset can be applied with no cost as long as the application server does not fault when failing to send request metadata to the limiting server.

References

[1] Raffi Krikorian. New tweets per second record, and how! *Link*, August 2013.

[2] Ed Erhart. “Dare to be different, yet hold your head high”: the impact of Princes death on Wikipedia. *Link*, April 2016.

[3] AWS Team. Summary of the Amazon DynamoDB service disruption and related impacts in the US-east region. *Link*, September 2016.

[4] AWS Team. Summary of the Amazon EC2, amazon EBS, and Amazon RDS service event in the EU west region. *Link*, August 2016.

[5] AWS Team. Summary of the Amazon EC2 and Amazon RDS service disruption in the US east region. *Link*, April 2011.

[6] Salvatore Sanfilippo. Redis - in-memory data structure store. *Link*, May 2016.

[7] Nikrad Mahdi. An alternative approach to rate limiting. *Link*, April 2017.

[8] Robert Paprocki. How to design a scalable rate limiting algorithm. *Link*, December 2017.

[9] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC’10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[10] Flavio P. Junqueira and Bejnamin Reed. *ZooKeeper: Distributed Process Coordination*. O’Reilly Media, 2013.

[11] Peter Bailis and Kyle Kingsbury. The network is reliable. *Queue*, 12(7):20:20–20:32, July 2014.

[12] Arnon Rotem-Gal-Oz. Fallacies of distributed computing explained. *Link*, 2006.

[13] Google Inc. YouTube/Doorman: Global distributed client side rate limiting. *Link*, 2016.

[14] Jose Nino. Announcing Ratelimit : Go/gRPC service for generic rate limiting. *Link*, February 2017.

[15] Varad Kishore. Cloud Bouncer - distributed rate limiting at Yahoo. *Link*, February 2015.

[16] Tim Starling. PoolCounter. *Link*, May 2017.

[17] Andrew Rodland. Smockron. *Link*, December 2017.

[18] Paul Tarjan. Scaling your API with rate limiters. *Link*, March 2017.