Rate Limit Refinements on Stream Types

LUCAS DU, University of California, Davis, USA
CALEB STANFORD, University of California, Davis, USA

Abstract

1 Introduction

2 Rate Limiters in the Real World

There are a number of well-understood algorithms that are used in practice to implement rate limiters. These implementation strategies can broadly be categorized as either A) "window"-based or B) "bucket"-based. Furthermore, each of these strategies guarantees a particular *event*[1] *distribution* (or *traffic shape*, from the networks literature).

In this section, we briefly survey different kinds of real-world rate limiter implementations and discuss the event distributions they impose. We then classify these event distributions, grouping specific implementations by core commonalities, and use this classification to inform the design of our type system.

2.1 Rate Limiter Implementation

Rate limiter implementation broadly falls into four categories: (1) fixed window, (2) sliding window, (3) token bucket, and (4) leaky bucket. Here, we briefly outline the details of their implementation necessary to understand their corresponding event distributions. We also briefly discuss some of their practical benefits and drawbacks.

2.1.1 Fixed Window. In a fixed window implementation, time is divided into fixed windows (hence the name) and limits are enforced within those fixed windows using a counter that is reset at the end of every window.

Fixed window implementations are usually very efficient, since they only require a single counter to be incremented and checked. However, they are vulnerable to bursts of traffic, particularly at window edges. For example, if we enforce a rate limit of 10 requests per minute using a fixed window approach, 10 requests could come in at the very end of one window and 10 more could come in at the start of the next window—we have now allowed 20 requests in a very short span of time, possibly overwhelming our system.

One common approach to alleviating this "burstiness" is to use multiple fixed window rate limits simultaneously, often by combining a limit with a large window and a limit with a smaller window. For example, we could combine our 10 requests per minute rate limit with a 3 requests per 5 seconds rate limit: the addition of the smaller window limit smooths out bursts that were previously possible (i.e. around the edges of the 1 minute windows, we now only allow 6 requests instead of 20). See Section 3.5 for a discussion of how this can be handled by the type system.

[1]In different contexts, the measure of data being rate limited might be "events", "requests", bits in incoming packets, and so on. Each of these has a slightly different connotation, but they all ultimately serve the same purpose: to act as a consistent unit of measure. We will use the term "events" for simplicity and consistency.

August 9, 2025.
Authors' Contact Information: Lucas Du, lzdu@ucdavis.edu, University of California, Davis, USA; Caleb Stanford, cdstanford@ucdavis.edu, University of California, Davis, USA.

2.1.2 Sliding Window. A sliding window implementation also enforces specific limits within windows, but instead of setting fixed windows, it uses a "sliding" window of size t that moves with incoming events. For a sliding window of size 10 seconds, for example, we would take the number of events in the *last* 10 seconds before the latest incoming event and check if the total is at our specified limit. If it is under the limit, we allow the latest event to pass through; otherwise, we reject the event.

2.1.3 Token Bucket. A token bucket algorithm (which is equivalent to a flipped version of the "leaky bucket as a meter" algorithm) rests on the notion of a bucket of some depth d that is continuously being filled at some specified rate by tokens—the bucket cannot be filled past its maximum depth d and tokens are consumed and removed from the bucket by incoming events. Incoming events are only passed through if there is a token to consume, i.e. if an event comes in and there are no tokens in the bucket, that event is rejected.

This approach is able to handle bursts of traffic—the bucket acts as a buffer and the depth of the bucket is the maximum burst that is allowed—but the average rate of events is the same as the rate at which tokens are being refilled.

2.1.4 Leaky Bucket. The leaky bucket algorithm has two interpretations, largely due to historical accident: the "leaky bucket as a meter" interpretation and the "leaky bucket as a queue" interpretation.

The "as a meter" interpretation is equivalent to the token bucket approach (more specifically, a flipped version of the token bucket: in the "leaky bucket as a meter," incoming events are *added* as tokens to the bucket rather than *consumed* and the bucket is continuously being *drained* at some specified rate instead of being *filled*), so we focus here on the "as a queue" interpretation.

In the "leaky bucket as a queue," incoming events fill the bucket with tokens. If the bucket is full, incoming events are rejected. A "leak" in the bucket continuously drains the bucket at some specified rate, allowing events to pass through. This bucket here is essentially acting as a queue (hence the name) and the "leak" directly imposes a smooth rate limit: events can only pass through at the rate of the leak, so there are no bursts.

2.2 Classifying Event Distributions

We can broadly classify the types of event distributions these implementations impose into two semantically distinct classes: *segmented* and *uniform*. These are defined in terms of windows, but also map onto the kinds of distributions created by the "bucket"-based algorithms.

A *segmented* distribution is one in which time is divided into fixed windows (of size t) and event counts are capped within each of those fixed windows. A *uniform* distribution is one in which event counts are capped within *for any* window of size t.

Here is the mapping of rate limiting algorithms onto these classes of event distributions:

- The fixed window algorithm creates a segmented distribution, essentially by definition.
- The sliding window algorithm creates a uniform distribution, also essentially by definition. Specifically, the formulation of the algorithm enforces a limit of n events over *any* span of t time.
- The token bucket algorithm is a bit trickier, but it can be modeled as a uniform distribution. The trickiness arises due to the way that the token bucket allows bursts up to the depth d of

the bucket, but still imposes an *average* rate that matches the rate of token replenishment. Still, we are imposing limits *for any* window of t time, not just fixed segments of time. [2]

- The "leaky bucket as a queue" algorithm creates a uniform distribution. The rate limit corresponds to the rate of the "leak," which applies *for any* window of time. In fact, given a leak rate n/t, the leaky bucket (as a queue) algorithm guarantees that the rate is either *exactly* n/t for any window of size t or simply 0.

3 Subtyping Rules

3.1 Basic Rules

S-RATE is the rule for subtyping between stream types with rate limit refinements and underlying "bare" stream types. S-RELATIVE-RATE is the rule for subtyping between two stream types with rate limit refinements with equal time windows. Note that these hold for both uniform and segmented rate distributions; for simplicity, we just use the uniform rate notation. For completeness, we also list out the basic structural subtyping rules for each type constructor.

$$\textsc{s-rate}\quad \frac{}{S_{@n/t} <: S} \qquad \textsc{s-relative-rate}\quad \frac{n \leq m}{S_{@n/t} <: S_{@m/t}}$$

$$\textsc{s-concat}\quad \frac{S_1 <: S_2 \qquad T_1 <: T_2}{S_1 \bullet T_1 <: S_2 \bullet T_2} \qquad \textsc{s-sum}\quad \frac{S_1 <: S_2 \qquad T_1 <: T_2}{S_1 + T_1 <: S_2 + T_2} \qquad \textsc{s-par}\quad \frac{S_1 <: S_2 \qquad T_1 <: T_2}{S_1 \parallel T_1 <: S_2 \parallel T_2} \qquad \textsc{s-star}\quad \frac{S <: T}{S^* <: T^*}$$

NOTE: I think the S-CONCAT rule might be messed up. To guarantee semantic safety (yet to be proven, of course), we might need it to replace the S_2 and T_2 supertypes in the premise with some $U = \text{glb}(S_2, T_2)$.

3.2 Uniform Distributions

$$\textsc{s-uniform-concat-expand}\quad \frac{}{(S \bullet T)_{@n/t} <: S_{@n/t} \bullet T_{@n/t}} \qquad \textsc{s-uniform-concat-factor}\quad \frac{}{S_{@n_1/t} \bullet T_{@n_2/t} <: (S \bullet T)_{@(n_1+n_2)/t}}$$

$$\textsc{s-uniform-sum-expand}\quad \frac{}{(S + T)_{@n/t} <: S_{@n/t} + T_{@n/t}} \qquad \textsc{s-uniform-sum-factor}\quad \frac{}{S_{@n_1/t} + T_{@n_2/t} <: (S + T)_{@\max(n_1,n_2)/t}}$$

$$\textsc{s-uniform-par-expand}\quad \frac{}{(S \parallel T)_{@n/t} <: S_{@n/t} \parallel T_{@n/t}} \qquad \textsc{s-uniform-par-factor}\quad \frac{}{S_{@n_1/t} \parallel T_{@n_2/t} <: (S \parallel T)_{@n_1+n_2/t}}$$

$$\textsc{s-uniform-star-expand}\quad \frac{}{(S^*)_{@n/t} <: (S_{@n/t})^*} \qquad \textsc{s-uniform-star-factor}\quad \frac{}{(S_{@n/t})^* <: (S^*)_{@2n/t}}$$

As a note, S-UNIFORM-STAR-FACTOR requires a semantic property of the *length* of streams $S_{@n/t}$ to hold: such streams *must complete a window* (of length t). More concretely, the length of such a stream must be $\geq t$.

NOTE: I think concatenation also needs this "complete window" requirement, since concatenations can be composed multiple times and thus suffers the same problem as star/iteration in terms of not knowing how many different streams could be in one crosssover period.

[2]There is a temptation here to classify this as a semantically distinct third distribution class, i.e. a *bursty* distribution. However, it is possible to precisely model this distribution as a uniform distribution—the intuition is essentially as described here, but the details will come in Section 3.6

NOTE: I think to really get this to be useful as a typechecker, we also need to have some typing rules convert between type constructors. To do this, I believe we need some base type like Int. Particularly, I think the simplest/most naive solution here (just so we can get a basic subtyping checker to work) is to have rules that have Int as a subtype of all Stream Types constructors, i.e. Int@(r0) is subtype of Concat(Int@r1, Int@r2) when r0 is "less than" (i.e. is a subtype of) r1, r2 in the rate refinement typing lattice. This could probably be made more general, actually, but I'm just going to think of things onlly as Ints for now. I think the rules for the other constructors fairly straightforward, i.e. for Sum, it's just the lesser of the two types being summed, and for Par it's just the sum of the two types. Not sure if all of this will scale when we have more types that possibly aren't compatible, but we'll see. We'll also see if this makes sense later on, or if it just seems like I'm rambling.

3.3 Segmented Distributions

S-SEGMENT-CONCAT-EXPAND

$$\frac{}{(S \bullet T)_{|@n/t|} <: S_{|@n/t|} \bullet T_{|@2n/t|}}$$

S-SEGMENT-CONCAT-FACTOR

$$\frac{}{S_{|@n_1/t|} \bullet T_{|@n_2/t|} <: (S \bullet T)_{|@\max(n_1+n_2,2n_2)/t|}}$$

S-SEGMENT-SUM-EXPAND

$$\frac{}{(S + T)_{|@n/t|} <: S_{|@n/t|} + T_{|@n/t|}}$$

S-SEGMENT-SUM-FACTOR

$$\frac{}{S_{|@n_1/t|} + S_{|@n_2/t|} <: (S + T)_{|@\max(n_1,n_2)/t|}}$$

S-SEGMENT-PAR-EXPAND

$$\frac{}{(S \parallel T)_{|@n/t|} <: S_{|@n/t|} \parallel T_{|@n/t|}}$$

S-SEGMENT-PAR-FACTOR-1

$$\frac{(S \parallel T) \text{ align } S \text{ align } T}{S_{|@n_1/t|} \parallel T_{|@n_2/t|} <: (S \parallel T)_{|@(n_1+n_2)/t|}}$$

S-SEGMENT-PAR-FACTOR-2

$$\frac{(S \parallel T) \text{ align } S}{S_{|@n_1/t|} \parallel T_{|@n_2/t|} <: (S \parallel T)_{|@(n_1+2n_2)/t|}}$$

S-SEGMENT-PAR-FACTOR-2

$$\frac{(S \parallel T) \text{ align } T}{S_{|@n_1/t|} \parallel T_{|@n_2/t|} <: (S \parallel T)_{|@(2n_1+n_2)/t|}}$$

S-SEGMENT-STAR-EXPAND

$$\frac{}{(S^*)_{|@n/t|} <: (S_{|@2n/t|})^*}$$

S-SEGMENT-STAR-FACTOR

$$\frac{}{(S_{|@n/t|})^* <: (S^*)_{|@2n/t|}}$$

In S-SEGMENT-STAR-EXPAND, the factor of 2 in the supertype comes from the fact that a window of a substream type S in the supertype can overlap at most 2 windows of stream type S^* in the subtype. Thus, to enforce subset semantics, we must allow up to $2n$ events in a window for substream type S.

As in the uniform version of these rules, S-SEGMENT-STAR-FACTOR depends on the requirement that a stream of type $S_{@n/t}$ *must complete a window*: in other words, the length of such a stream must be $\geq t$. This guarantees that there will never be more than 2 windows of the substream type S in the subtype overlapping a window of stream type S^* in the supertype.

3.4 Conversions Between Distributions

S-SEGMENT-OF-UNIFORM

$$\frac{}{S_{@n/t} <: S_{|@n/t|}}$$

S-UNIFORM-OF-SEGMENT

$$\frac{}{S_{|@n/t|} <: S_{@2n/t}}$$

3.5 Different Window Sizes

So far, we have only considered subtyping relations between refinements where event counts change, but the window size t remains the same. What happens to subtyping relations if we allow the window sizes to be different? There are two cases: either the subtype has a larger window size or a smaller one.

First, we lay out the rules when the subtype has a larger window size.

S-UNIFORM-SUBWINDOW-LARGER
$$\frac{t_2 \leq t_1}{S_{\mathsf{@n/t_1}} <: S_{\mathsf{@n/t_2}}}$$

S-SEGMENT-SUBWINDOW-LARGER-DIVISIBLE
$$\frac{t_2 \leq t_1 \qquad t_1/t_2 \in \mathbb{N}}{S_{\mathsf{|@n/t_1|}} <: S_{\mathsf{|@n/t_2|}}}$$

S-SEGMENT-SUBWINDOW-LARGER
$$\frac{t_2 \leq t_1}{S_{\mathsf{|@n/t_1|}} <: S_{\mathsf{|@2n/t_2|}}}$$

Next, we lay out the rules when the subtype has a smaller window size. To make these work, we need to impose more nuanced constraints on the relative event counts in a way that corresponds to the relative window sizes.

S-UNIFORM-SUBWINDOW-SMALLER
$$\frac{t_1 \leq t_2 \qquad n_1 \leq n_2/(\lceil t_2/t_1 \rceil)}{S_{\mathsf{@n_1/t_1}} <: S_{\mathsf{@n_2/t_2}}}$$

S-SEGMENT-SUBWINDOW-SMALLER-1
$$\frac{t_1 \leq t_2 \qquad (t_2 - (\lfloor t_2/t_1 \rfloor \cdot t_1)) \leq g \qquad n_1 \leq n_2/(\lceil t_2/t_1 \rceil)}{S_{\mathsf{@n_1/t_1}} <: S_{\mathsf{|@n_2/t_2|}}}$$

S-SEGMENT-SUBWINDOW-SMALLER-2
$$\frac{t_1 \leq t_2 \qquad (t_2 - (\lfloor t_2/t_1 \rfloor \cdot t_1)) > g \qquad n_1 \leq n_2/(\lceil t_2/t_1 \rceil + 1)}{S_{\mathsf{@n_1/t_1}} <: S_{\mathsf{|@n_2/t_2|}}}$$

In the segmented rules, g represents the *greatest common time granularity* of t_1 and t_2. For example, if t_1 is 2 second and t_2 is 5 seconds, g would be 1 second; if t_1 is 0.3 seconds and t_2 is 4 seconds, g would be 0.1 seconds; if t_1 is 3 seconds and t_2 is 4.55 seconds, g would be 0.01 seconds.

NOTE: I actually think now that it's easier to think about all this if we *combine* our reasoning about rate refinements into a single framework. Instead of thinking about what happens with same window size+different events and then thinking separately about different window sizes, we should just treat rate refinements as always possibly having different events *and* window sizes and having rules that handle both simulateously. It probably makes things easier to understand and also makes implementing the subtyping checking more straightforward.

3.6 Modeling Bursty Distributions

We can model *bursty* distributions (of the kind produced by token bucket limiters) precisely as uniform distributions. Specifically, the bursty distribution allows a maximum burst of d events (where d is the depth of the bucket), but otherwise enforces a rate of n/t (n events per t time). **TODO:** Figure out the details here.

3.7 Refinement Casting and Multiple Refinements

We can also apply multiple rate limit refinements to a stream type. **TODO:** There are two approaches to this—the simplest is to treat this as a *cast* and simply take the strongest refinement (as given by the subtyping rules) as the refinement. However, this leaves two problems:

(1) What do we do if there is no subtyping relation between all of the refinements applied? If we treat multiple refinements as a cast, the simplest answer seems to be to just disallow these kinds of refinements.
(2) There may be use in allowing *conjunctions* of refinements (even ones that have no subtyping relationship), so erasing that information may be problematic. For example, a common strategy to control burstiness when using fixed window rate limiting is to have a have 2 limits, i.e. 10 events per 1 minute, then a burstiness control limit of 2 events per 1 second. There is no subtyping relationship here (at least given our current rules), but this should still be allowed as it expresses an important constraint.

NOTE: I actually don't think these two interpretations of multiple rate refinements are contradictory. We can treat multiple refinements as logical conjunctions, and also erase a refinement if it is a supertype of an existing refinement on the stream type. The erasure of the supertype is not at odds with the semantics of logical conjunction and the implication relation that corresponds to subtyping; intuitively, if we already have the stronger version of some logical predicate, we don't need the weaker version in order to get the same power of implication.

TODO: We also need to try and get **more precise constraints** on currently problematic types like (a) "bursty" streams and (b) the forced $n_1 + n_2$ or $2n$ things to handle crossover points in concatenated/star streams.

- It seems like using larger window sizes can help us get a handle on "bursty streams" produced by token bucket or "leaky bucket as meter" rate limiters (i.e. something like: if bucket depth is 10 and rate of replenishment is 2/1sec, instead of being forced to have a consistent uniform rate of (10+2)/1sec, we could instead have something like $(10 + 2 \cdot 5 = 20)$/5sec...the math here needs to be worked out some more).
- It also seems possible that working out the multiple refinements thing, which is ostensibly for modeling certain fixed window strategies, could also help give tighter bounds for both "bursty" streams and the annoying $n_1 + n_2$ sort of things.

NOTE: I don't think multiple refinements in the way we are imagining them above (as logical conjunctions) will really help with the annoying addition stuff we need to do currently for concatenation (and relatedly, iteration). The main idea for this is basically to have two refinements: one with the larger (or smaller, depending on if we want a supertype or a subtype) and the other is the rate limit over the crossover windows, i.e. $(n_1 + n2)/(t_1 + t_2)$. Maybe this does work? It does seem to be a sound overapproximation (at least semantically) of the concatenated stream...and is a bit tighter than simply doing addition? Also, if we use a unified framework for rate refinements (combining different event numbers *and* window sizes), then this actually seems like the only reasonable way forward. I don't know; this deserves more thought. As does handling "bursty streams." FWIW, the idea with fixing "bursty streams" with multiple refinements is similar.

NOTE: Also, for concatenated streams, it appears that we may need to extend the rate type refinement to 3 places. Specifically: `(entry refinement | min crossover | exit refinement)`. Otherwise, it seems that composition of concatenated types loses *a lot* of precision, since we'll just keep on adding max rates together without any notion of what crossovers are actually possible. Hopefully this makes sense (and writing out an example should be illuminating).

References