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.

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 an incoming event. 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:
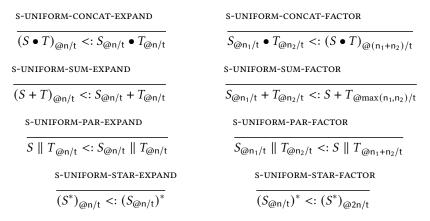
- The fixed window algorithm creates a segmented distribution, essentially by definition.
- The sliding window algorithm creates a uniform distribution, also essentially by definition. Specifically, the formulation of the algorithm enforces a limit of n events over *any* span of t time.
- The token bucket 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.
- 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.

TODO: Perhaps we can add a third "bursty" class, although the current approach of just modeling it using the uniform class might be good enough.
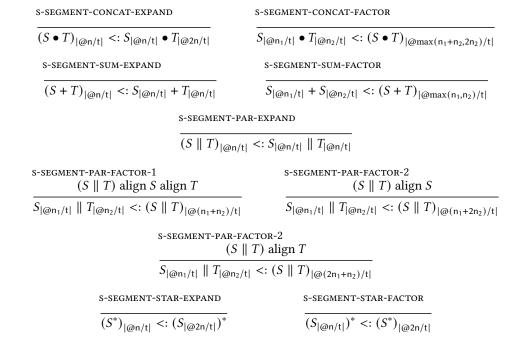
3 Subtyping Rules

3.1 Uniform Distributions

S-UNIFORM-CONCAT-EXPAND
$$(S \bullet T)_{@n/t} <: S_{@n/t} \bullet T_{@n/t}$$

S-UNIFORM-CONCAT-FACTOR
$$S_{@n_1/t} \bullet T_{@n_2/t} <: (S \bullet T)_{@(n_1+n_2)/t}$$

S-UNIFORM-SUM-EXPAND
$$(S + T)_{@n/t} <: S_{@n/t} + T_{@n/t}$$

S-UNIFORM-SUM-FACTOR
$$S_{@n_1/t} + T_{@n_2/t} <: S + T_{@\max(n_1,n_2)/t}$$

S-UNIFORM-PAR-EXPAND
$$S \parallel T_{@n/t} <: S_{@n/t} \parallel T_{@n/t}$$

S-UNIFORM-PAR-FACTOR
$$S_{@n_1/t} \parallel T_{@n_2/t} <: S \parallel T_{@n_1+n_2/t}$$

S-UNIFORM-STAR-EXPAND
$$(S^*)_{@n/t} <: (S_{@n/t})^*$$

S-UNIFORM-STAR-FACTOR
$$(S_{@n/t})^* <: (S^*)_{@2n/t}$$

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 $\geq t$.

3.2 Segmented Distributions

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

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

S-SEGMENT-SUM-EXPAND
$$(S + T)_{|@n/t|} <: S_{|@n/t|} + T_{|@n/t|}$$

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

S-SEGMENT-PAR-EXPAND
$$(S \parallel T)_{|@n/t|} <: S_{|@n/t|} \parallel T_{|@n/t|}$$

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

S-SEGMENT-STAR-FACTOR
$$(S_{|@n/t|})^* <: (S^*)_{|@2n/t|}$$

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.

3.3 Conversions Between Distributions

S-SEGMENT-OF-UNIFORM

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

S-UNIFORM-OF-SEGMENT

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

3.4 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

$$\frac{t_2 \leq t_1}{S_{@n/t_1} <: S_{@n/t_2}}$$

S-SEGMENT-SUBWINDOW-LARGER-DIVISIBLE

$$\frac{t_2 \leq t_1 \qquad t_1/t_2 \in \mathbb{N}}{S_{|@n/t_1|} <: S_{|@n/t_2|}}$$

S-SEGMENT-SUBWINDOW-LARGER

$$\frac{t_2 \leq t_1}{S_{|@n/t_1|} <: S_{|@2n/t_2|}}$$

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

$$\frac{t_1 \leq t_2 \qquad n_1 \leq n_2/(\lceil t_2/t_1 \rceil)}{S_{@n_1/t_1} <: S_{@n_2/t_2}}$$

S-SEGMENT-SUBWINDOW-SMALLER-1

$$\frac{t_1 \leq t_2 \qquad (t_2 - (\lfloor t_2/t_1 \rfloor \cdot t_1)) \leq g \qquad n_1 \leq n_2/(\lceil t_2/t_1 \rceil)}{S_{@n_1/t_1} <: S_{@n_2/t_2}}$$

S-SEGMENT-SUBWINDOW-SMALLER-2

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

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.

3.5 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.

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 mutliple 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.

References