Distributed Rate Limiter

Matthew Torre Amy Chang Andy Wang

Abstract

We re-implement *Cloud Control with Distributed Rate Limiting* by Raghavan et al. (SIGCOMM 2007), which introduces a distributed system for enforcing a global network rate limit across geographically distributed sites. Distributed rate limiting is useful when a centralized limiter is infeasible or introduces unacceptable latency. The paper proposes two algorithms: Global Random Drop (GRD), which estimates global load via distributed aggregation and probabilistically drops packets when demand exceeds the limit, and Flow Proportional Share (FPS), which uses TCP-aware flow weighting to preserve fairness across competing flows. Our work focuses on replicating key experimental results from the original paper, particularly demonstrating that distributed algorithms converge to the behavior of a centralized token bucket and maintain fair capacity allocation under skewed flow distributions.

1 Introduction

1.1 Background

Cloud Control with Distributed Rate Limiting is a pioneering paper that introduced the idea of distributed rate limiting. The paper presents distributed rate limiting in the context of allowing service providers to exercise greater control over the ingress traffic into their service, when multiple servers are geographically disjoint and separated by a 40ms round-trip time (RTT).

At the time, the sole alternative used local rate limiters at each server instance. This fails in distributed deployments. If the local limit is set too high, such as at the global rate limit, then the client can access multiple different servers to bypass the global rate limit. If the local limit is set lower than the global rate limit, then the client cannot achieve the advertised rate if only communicating with a single server. A distributed rate limiter maintains the global rate limit regardless of which servers are accessed.

The paper introduces two distributed rate limiter designs specifically for TCP: Global Random Drop, and Flow Proportional Share. GRD drops incoming packets at a rate proportional to how much the ingress rate exceeds the global limit, and FPS is an enhanced token bucket-based algorithm where the local limit at each limiter depends on the weight calculation based on the states of other limiters [1].

1.2 Scope of Replication

We reproduce Figure 3 of the paper. There are two rate limiters with 40 ms RTT between them, each running one of the four rate-limiting algorithms:

- Central Token Bucket
- Global Token Bucket
- Global Random Drop
- Flow Proportional Share

In this experiment, 7 TCP flows traverse limiter 1 and 3 TCP flows traverse limiter 2. We aim to show that each algorithm approximately converges to the global rate limit, while differing in exact convergence behavior and fairness.

2 Rate Limiters

2.1 Central Token Bucket (CTB)

Central Token Bucket is the benchmark rate-limiting mechanism for a single rate limiter. Tokens are continuously added to a bucket at a fixed rate - the target rate. Arriving packets attempt to consume tokens equal to the size of the packet - if there are not enough tokens, then the packet is dropped. A sufficiently large bucket allows bursts to be handled while respecting the long-term target rate.

2.2 Global Token Bucket (GTB)

Global Token Bucket is mentioned but not described in the paper due to fairness concerns, as we will demonstrate in section 4. This method maintains a token bucket and adds tokens at the global rate. Further, it assumes each limiter knows the ingress rates at other limiters, and decrements its own token bucket by that amount to account for traffic through other limiters. Similar to CTB, when a packet arrives on a rate limiter, if there are enough tokens then the packet will be allowed through; otherwise the packet will be dropped.

2.3 Global Random Drop (GRD)

Similar to GTB, Global Random Drop requires an estimate of the aggregate global demand across all flows. However, instead of a token bucket, each limiter calculates the packet drop probability as $\frac{(demand-limit)}{demand}$ and drops each incoming packet with this probability. If the global demand is less than the rate limit, it does not proactively drop packets.

2.4 Flow Proportional Share (FPS)

FPS is designed to allocate capacity according to TCP flow demand across multiple flows. Regulating traffic purely based on aggregate demand may result in unfair allocation, as we will show for GTB. Instead, FPS allocates capacity to each rate limiter based on the inferred flow demand at each limiter, accounting for factors such as the largest unbottlenecked flow rate.

Similar to GTB, FPS maintains a local token bucket. However, instead of gossiping an estimated global rate, each limiter estimates a local weight that reflects its share of global demand, which is then gossiped periodically between limiters. Using the aggregated weights, each limiter then computes a rate limit proportional to its share of the global weight for the token bucket. Implementation-specific details are presented in section 3.

3 System Design and Implementation

3.1 Architecture Overview

We have implemented all the components required to reproduce this paper in C++. There are three distinct processes - traffic senders send TCP packets at a fixed or uncapped rate, traffic relays run the rate limiting algorithm and forward data from the sender to receiver, and a single traffic receiver that is responsible for collecting final throughput statistics from each flow.

Each traffic sender sends data with the relay as the destination, and the relay forwards that data to the traffic receiver using UDP. In the original paper, each relay forwards the packets to the destination using a raw socket - our implementation is simpler while preserving the same rate-limiting behavior for our experiments. The relay process uses `NFQUEUE` to process packets. Admitted packets receive an `NF_ACCEPT` verdict, after which the kernel sends a normal ACK to the sender. Dropped packets receive an `NF_DROP` verdict and are silently discarded; no acknowledgement is generated. Because TCP uses AIMD congestion control, the missing ACK eventually triggers a timeout and window reduction.

Not only does the rate limiting decision have to occur within the `NFQUEUE` callback, both the local and the global rates used for limiting decisions must be measured here before the drop occurs. The paper alludes to this by saying that limiters "depend upon the local packet arrival rates". Convergence of the egress rate to the global limit is a byproduct; the rate limiters instead measure ingress rate, which more accurately reflects instantaneous demand.

3.2 Gossip Coordination

In the paper, gossip messages are sent between traffic relays using UDP datagrams with 20 bytes of payload, and the primary goal is to compute and distribute the sum of all ingress rates at the limiters, as well as the weight for FPS.

Peer discovery is a major component of gossiping, since knowing how many peers are online ensures conservation of mass and that data is sent to the correct recipients. Because the paper is not concerned with gossiping, the authors gloss over such implementation details. We employ a Redis database containing a registry of all relays, and each relay periodically polls this database to locate peers.

3.3 FPS-Specific Implementation

In this section we describe the details of our FPS implementation, which is the most complex rate limiting algorithm. Every 100 ms, each relay recomputes its ideal weight and local limit. For each active flow, the relay computes the interval byte rate and updates its EWMA using $\alpha = 0.2$:

$$\hat{r}_f \leftarrow 0.2\,\frac{B_f}{\Delta t} + 0.8\,\hat{r}_f,$$

where B_f is the number of bytes observed during interval Δt. Flows idle for more than 3 seconds are expired. Let d denote the local ingress rate, L the current local limit, and r^* the maximum per-flow EWMA rate. If the relay is at capacity ($d \geq L$), it sets its raw ideal weight to $w^* = L/r^*$, which approximates the number of equally sized flows it is serving. If the relay is below capacity ($d < L$), it instead sets

$$w^* = \frac{d \cdot W_{remote}}{G - d},$$

where W_{remote} is the sum of peer weights received through gossip and G is the global limit. This raw weight is clamped to $[0.1, 1000]$ and smoothed with an EWMA using $\alpha = 0.2$. The relay then updates its share of the global limit as

$$L_{new} = \frac{w}{w + W_{remote}} \cdot G,$$

applies a second EWMA with $\alpha = 0.2$, and adopts the result as its new local limit before gossiping the updated weight to its peers. This local limit is used when updating the rate limiter's token bucket.

3.4 Simulation Environment

In order to reproduce the exact environment from the paper, all TCP sources use New Reno with SACK enabled, as specified in the paper.

We use Mininet to emulate network latency between different hosts and manage all the processes on a single machine. In the paper, both the source-sink and inter-limiter RTT are 40ms. The experiment is sensitive to source-sink latency because low latency, such as using the loopback interface without delay, will induce heavy oscillation due to TCP ramping up and down quickly in response to drops.

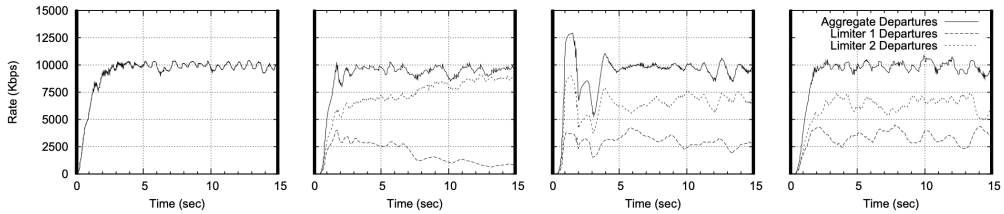
Another benefit of Mininet is that traffic between different hosts uses 1500-byte Ethernet frames. `NFQUEUE` decisions are made within the network layer, meaning that a drop decision will drop the entire frame. Our initial testing was done over loopback which uses 65536-byte packets, leading to inconsistent rates as packet drop decisions are less granular. Although we can set the loopback MTU to be 1500 bytes, Mininet's convenience makes it a clear choice of platform to run simulation in.

4 Evaluation

We aim to reproduce Figure 3 from the *Cloud Control* paper, which we reproduce as Figure 1 in our paper. This figure contains a time series of forwarding rates over 15 seconds for the 4 rate limiters evaluated in the paper. The data is gathered by setting a 10Mbps global rate limit across two relays, and sending seven unbottlenecked flows to relay 1 and three unbottlenecked flows to relay 2. By sending different numbers of flows to each relay, the paper demonstrates the fairness characteristics of each algorithm. We do not discuss CTB in detail, since it serves primarily as a baseline.

4.1 Global Token Bucket replication

GTB exhibits severe unfairness under the 7–3 flow split. While it enforces the global rate limit, the limiter serving 7 flows consistently dominates the limiter serving 3 flows. We observed this



(a) Centralized token bucket. (b) Global token bucket. (c) Global random drop. (d) Flow proportional share.

Figure 1: Figure 3 from the *Cloud Control* paper - Time series of forwarding rate for various rate limiter designs

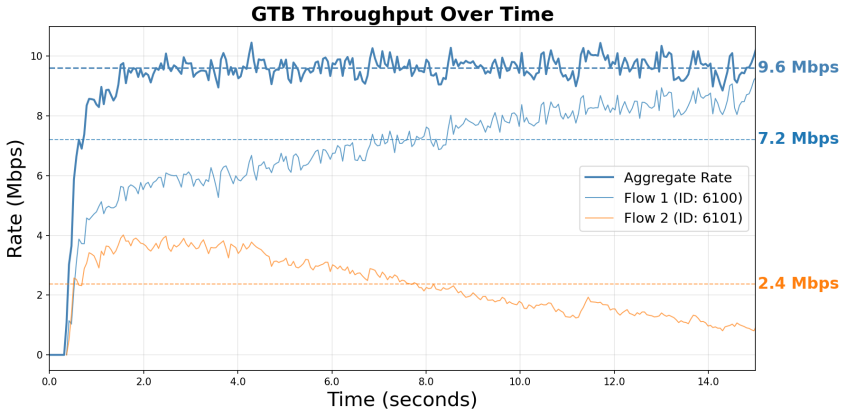


Figure 2: Reproduced forwarding rate for Global Token Bucket

trend in Figure 1(b) of the *Cloud Control* paper and are able to reproduce the same behavior in our implementation in Figure 2, where the 3-flow limiter makes minimal forward progress.

We hypothesize that the unfairness stems from the dependence of each limiter's token refill rate on estimated ingress rates at other limiters. Because the 7-flow limiter observes a higher ingress rate, it can exploit available slack more quickly and advertise a higher local demand through gossip, thereby capturing a disproportionate share of the global capacity. In contrast, the 3-flow limiter can only slow down and drop more packets when it observes that the rates at other relays are high. It cannot detect that the other limiter is monopolizing throughput. Figure 1(b) illustrates this effect: near 7 seconds there is noticeable slack in the global throughput, and the forwarding rate of the 3-flow limiter dropped significantly after this time as the 7-flow relay occupied more global throughput.

On the whole, GTB has poor fairness that makes it infeasible for real-world deployment.

4.2 Global Random Drop replication

GRD produces more oscillations and overshoot in the global rate than the other algorithms. In Figure 1, the global rate sharply climbs to roughly 13Mbps before overcorrecting to 5Mbps. Although the global rate converges to the target 10Mbps, large oscillations still periodically occur.

We observed the same overall behavior in our implementation, as shown in Figure 3, though the initial overshoot was even larger, peaking near 20Mbps. We faced difficulty reproducing the results of Global Random Drop using only the described algorithm - the rate oscillation was higher than what the paper shows and did not settle near the desired rate. Our conclusion was that packet drops were occurring too late. In the GRD algorithm's description, drops begin only after the estimated global rate exceeds the target limit. However, gossip is delayed by the 40ms RTT between relays, and the rate estimation itself from the paper uses an EWMA with $\alpha = 0.1$. This means that when the limiter sees the global rate at 10Mbps and starts to drop packets, the true instantaneous rate is likely even higher.

We hypothesize that the authors must have found ways to adjust the observed demand so it behaves closer to the actual demand. For our implementation, we apply another EWMA on the drop probability with $\alpha = 0.3$ to reduce sawtooth behavior, and begin to drop packets beyond 90% of the limit if

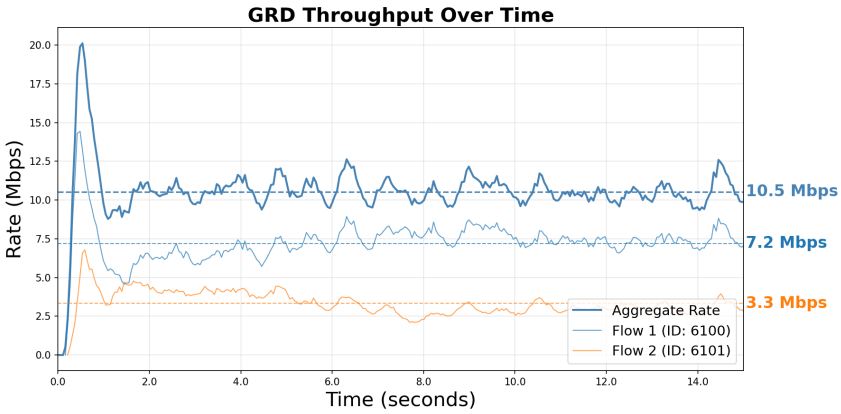


Figure 3: Reproduced forwarding rate for Global Random Drop

the global rate is increasing. Furthermore, we use a drop rate 50% more aggressive than stated by the paper when demand exceeds the global limit. All these modifications allow us to produce a graph similar to that presented in the paper. Together, they emulate the effects of observing a more up-to-date global rate estimation. Even with these changes, the achieved global rate is consistently around 10.4Mbps, which is likely because drops are still not happening promptly enough.

4.3 Flow Proportional Share replication

Figure 4 shows the forwarding rate for FPS. After an initial startup period where TCP flows ramp up their sending rates, the system converges to the expected proportional allocation. Relay 1 stabilizes near 6.2 Mbps while relay 2 stabilizes near 3.6 Mbps, and the aggregate throughput remains close to the 10 Mbps global limit.

Compared to global token bucket, FPS avoids the starvation behavior and maintains a balanced share of bandwidth between the two relays. This demonstrates that incorporating flow-aware weighting allows distributed limiters to approximate the fairness behavior of a centralized rate limiter while preserving the decentralized design.

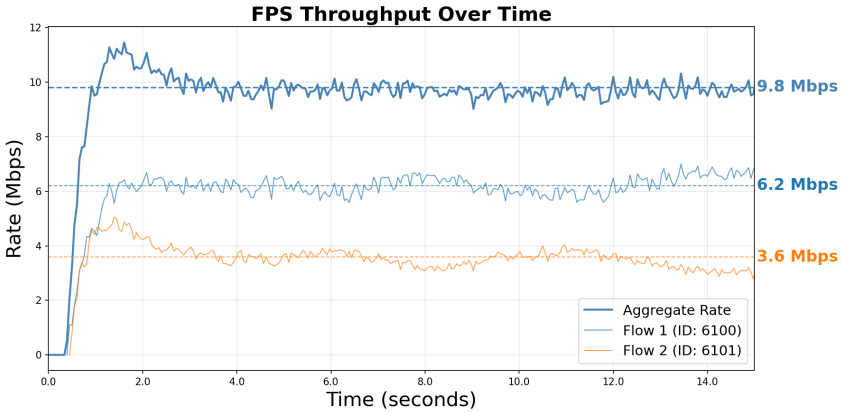


Figure 4: Reproduced forwarding rate for Flow Proportional Share

5 Discussion

5.1 Fairness between GRD and FPS

Comparing Figure 3 and 4, we can see that FPS exhibits slightly worse fairness than GRD. The 7-flow relay has a lower per-flow forwarding rate than the 3-flow relay in FPS, whereas in GRD the per-flow forwarding rates between the two relays are much closer. We can consistently reproduce this result. The *Cloud Control* paper also describes this effect - in Figure 5 of that paper, the authors show GRD

having higher Jain's fairness index than FPS. We hypothesize that this is because uniform dropping equalizes rates across all flows.

5.2 Scalability Considerations

The gossip protocol uses a branch factor of 2, sending to two random peers each round. In a push-sum-style setting, this gives roughly $O(\log N)$ convergence rounds for N relays. At 100 ms per round, 1,000 relays would require about 10 rounds, or roughly 1 second.

FPS stores $O(F)$ per-flow state per relay, where F is the number of active flows. The 3-second idle timeout bounds memory growth for short-lived traffic. Redis-based peer discovery is convenient for experiments but would need a more fault-tolerant replacement in production.

The EWMA coefficients used here were tuned for the two-relay case. Larger deployments with higher gossip latency may require smaller values, increasing convergence time.

6 Conclusion

We reproduced the main experimental results of Raghavan et al.'s distributed rate limiting paper [1], implementing CTB, GTB, GRD, and FPS algorithms in C++ under the original 7:3 sender split and 40 ms inter-relay RTT setting.

Our results support the paper's core claims. GTB enforces the global limit but allocates bandwidth unfairly, allowing the higher-flow relay to dominate. GRD achieves better per-flow fairness than FPS, nearly matching the centralized baseline. FPS produces the smoothest and most stable convergence, with an inter-limiter split closer to the ideal 7:3 target than GTB, though with slightly lower fairness than GRD.

The main engineering challenge was stabilizing GRD. Reproducing its behavior required extra smoothing and early-drop heuristics to compensate for lag in gossip-based load estimation. Overall, our implementation confirms that distributed rate limiting can approximate centralized fairness without a central coordinator.

References

[1] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '07)*, pages 337–348, Kyoto, Japan, 2007. ACM.

[2] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-Based Computation of Aggregate Information. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2003)*, pages 482–491, Cambridge, MA, USA, 2003. IEEE.

A Appendix

A.1 Use of AI

Our team has used AI assistant tools in the implementation of the project. We used AI to gain a deeper understanding of both the Cloud Control with Distributed Rate Limiting paper, as well as the push-sum gossip protocol. It helped us brainstorm reasons why we could not reproduce the results of the paper and identified bugs - for example, our initial implementation measured and gossiped local egress rate at the limiters, but AI recognized that measuring local ingress rate was more consistent with the paper's control objective than measuring local egress rate. It was also used for mundane tasks such as setting up the Mininet topology that produces the desired RTT and generating front-end visualizations.