Designing and Implementing a Fault Tolerant Distributed Rate Limiting Algorithm

Richard Ramsden
Directed Studies CSCI 485 under the supervision of Dr. David Wessels
Vancouver Island Unisveristy
In collaboration with Bravenet Media

ABSTRACT

Many cloud-based services need the ability to control traffic over multiple servers for Quality of Service (QoS) and pricing purposes. However, due to the distributed nature of the cloud enforcing a usage limit over multiple machines is non-trivial. In a distributed system servers fail for numerous reasons: power failures, broken hardware, etc. Worse, network problems occur frequently causing links between servers to break. Creating a distributed rate limting algorithm which works in the face of hardware failure and unreliable communication links is the main contribution in this paper. We provide a functional prototype written in Erlang to show how this algorithm can be used in a production environment

1. INTRODUCTION

Today, we are seeing more and more services switch to cloud-based architectures in order to scale with increasing demand, reduce costs, and improve availability. However, one of the biggest technical challenges with moving applications to the cloud is a loss of control. Many cloud-based services need the ability to control resource consumption for Quality of Service (QoS) and pricing purposes. For example, internet service providers need to limit individual download and upload speeds so they aren't overselling bandwidth. For QoS purposes: limiting resource consumption increases overall system availability since it ensures that one user cannot disrupt service for others.

That being said, the problem of controlling resource usage in the cloud is a non-trivial one. This is because it is difficult to achieve without using a centralized server. Large cloud-based companies such as Amazon seem to be finding it difficult to find a solution to this problem. For example, Amazon's Simple Storage Service (S3) allows customers to upload files to the cloud for hosting purposes. However, they have yet to find a solution for fine-grain control over user bandwidth[1]. This problem is not unique to Amazon, other content-delivery networks rely heavily on accurate rate limting algorithms to price out bandwidth for their clients.

In the first section of this paper, we formally define the distributed rate limiting (DRL) problem and present several algorithms for solving it. However, we demonstrate that existing algorithms have several problems: staleness and reliability on centralized servers.

In the second section, we propose a new approach expanding on existing research for solving the DRL problem. This section provides an algorithm for distributing a service rate over a cluster.

Lastly, we look at how the algorithm reacts to common network problems and server failures.

1.1 Problem Definition

The Distributed Rate Limiting (DRL) problem can be described as trying to distribute a service rate limit R, expressed as a number of service requests per time interval t, over a group of N servers in such a way that the supply S_i given to each server i is dependandant on the job population D_i at each server. For each timestep t our goal is to maintain:

$$\sum_{i=1}^{N} S_i = R$$

However, ensuring the aggregate sum of supply doesn't exceed the service rate is trivial. Maintaining a fair distribution of the service rate over time is the main challenge when designing a DRL algorithm.

1.2 A Trivial Approach

The easiest approach to the DRL problem is taking the *poorman's approach.* That is, taking a service rate R and dividing it evenly so supply at each server is equal. Thus, the distribution would be $S_1 = S_2 = ... = S_n = R/N$.

This approach works reasonably well when demand at each server is approximately equal. However, these conditions are rarely seen in production; more than likely each server will see varying demand. It is important to note that servers are not necessarily equal in terms of performance. A server with more ram and faster processing speed will process its job queue at a much faster rate.

At Bravenet Media we have implemented this approach in the past. However, in our old system requests were scheduled in a round-robin fashion which ensured that servers were given a fairshare over the entire job population. The next problem we encountered was that some servers could handle more demand than others causing the average number of processed requests to vary; a quick solution is to create virtual machines inside more performant servers. Of course, maintaining multiple virtual machines inside a server was troublesum and not the ideal solution for our team. More importantly, the real problem with this approach is that the DRL problem arises in services which are composed of geographically distributed sites[2]. Meaning demand can fluxuate depending on a number of factors, mainly latency.

1.3 Related Work

There have been several existing solutions to the DRL problem. The authors of *Cloud Control with Distributed Rate Limiting* proposed two algorithms for solving it: Flow Proportional Share (FPS) and Global Random Drop (GRD)[2]. These two algorithms are decentralized and use gossip protocols for sending updates to other nodes. Both, similar in design, have nodes periodically broadcasting their average number of service requests. Once a node has enough information on its neighbours it can estimate what its portion of the service rate should be. However, since limiters estimate their own supply without agreement from other limiters, overall demand can momentarily exceed the service rate. This happens because updates being sent around the cluster are stale and don't represent the instantaneous demand at a node. Similarly, Ajil, another DRL algorithm suffers from the same problem[4]. There is no "hard" limit instead most DRL algorithms have soft limits allowing demand to slightly fluxuate over the service rate. The only way to get around this problem is if supply can be passed around the cluster instead of being locally estimated. This is the basis of *Generalized Distributed Rate Limiting (GDRL)*[3].

GDRL is an analytic framework for the design of stable DRL algorithms. As mentioend prior, the basis of GDRL is that it works by passing supply around the cluster instead of having nodes rely on estimation. Thus, overall demand will never exceed the service rate making it the ideal framework for designing our DRL algorithm.

2. ALGORITHM DESIGN

As mentioned in the previous section we will use GDRL as the base to design our algorithm. However, first we need to formally define it.

2.1 The GDRL Framework

The goal of GDRL is to allocate supply in such a way that peformance at each server is uniform. Thus, in GDRL limiters work towards the fairness postulate:

Fairness postulate: The performance levels at different servers should be (aproximately) equal.

In order to formalize the fairness postulate we need the defintion of a *performance indicator*. Let q_i be the performance indicator at a server. The performance indicator can be anything (eg. "spare bandwidth", "mean response time"). Thus, the goal of GDRL is to ensure the following invariance is satisfied:

$$q_1 = q_2 = ... = q_N.$$

With the additional constraint:

$$\sum_{i=1}^{N} S_i = R$$

2.2 A New Approach

Now we present a modified version of the GDRL algorithm to solve the problem described in 1.1. For our performance indicator we will use "fill ratio" which we define as S_i/D_i. We want to satisfy the following:

$$\frac{S_1}{D_1} = \frac{S_2}{D_2} = ... = \frac{S_n}{D_n} \tag{1}$$

Satisfying this condition will ensure that every server receives a fairshare since the service rate is distributed depending on demand at a server. For communication, each server i will cooperate with its neighbours in a connected undirected graph.

Let there be N servers to control aggregate service rate R. The server i has a supply S_i that can be adjusted, and this server can exchange information with the server j if (i, j) is an edge in the communication graph $G = (N, E)$.

The first step of our algorithm will use the poorman's approach. Since we know nothing about the initial state of our cluster ie. how much demand is at each server; we will start by evenly dividing up the service rate.

The next step will allow us to achieve convergence (1). For each neighbour j of i we will take the difference between performance indicators at i and j and add that number to S_i to set the new supply.

$$S_i \leftarrow S_i + \eta \sum_{(i,j)\in E} (\frac{S_i}{D_i} - \frac{S_j}{D_j}) \tag{2}$$

We use the fraction η as a multiplier to ensure the stability of the algorithm. The authors of GDRL use $\eta = \frac{1}{2d}$ where d is the number of neighbours each node has. In our algorithm we will use $d = 2$ since using it is recommended in GDRL. If we didn't use η the algorithm would never converge since the values being passed around are too large.

2.3 Implementing the Aglorithm

Since $d = 2$ each node will have two neighbours. Periodically a node will query each of its neighbours asking for their fill ratio. Once obtaining a fill ratio it will take the difference between its ratio and its neighbour and multiply it by the fraction η. Let this value be $\delta d = \eta(\frac{S_i}{D_i} - \frac{S_j}{D_j})$. Looking at equation (2) we set the new supply at i with this value plus the old supply at i:

$$S_i \leftarrow S_i + \delta d$$

However, since were adding supply δd into S_i we can't create additional supply in our system since our original constraint would be broken. To fix this we subtract the additional supply from i's neighbour j. Thus, on the other end:

$$S_j \leftarrow S_j - \delta d$$

Repeating the above two steps for each node over time will achieve convergence.

For those interesteed, we have a prototype written in Erlang available online:

$$https://github.com/rramsden/experiments$$

3. FAULT TOLERANCE

The above algorithm is unique in the fact that it can survive hardware and link failures. In this section we will also discuss procedures for adding and removing nodes from the cluster.

3.1 Netsplits

A netsplit occurs when a link between servers is severed. A link can break for many reasons: node failure, routing issues, etc. When a netsplit does occur it segments a cluster into multiple clusters. Since our algorithm is decentralized it doesn't depend on a single master node. Nodes can keep working with their neighbours. If the netsplit seperates a node from its neighbour we take no action. We do this because servers will eventually rejoin the cluster since network issues will most likely eventually reslove themselves.

3.2 Node Failures

If a node goes offline for whatever reason: power failure, hardware failure, etc. a node will have a *best buddy node* which will inherit the failed nodes supply. When the node comes back online it will be passed supply based on the demand its seeing.

3.3 Adding and Removing Nodes

Adding new nodes can be handled by letting a node with no supply enter the cluster. The node will then pick any two random nodes to be its neighbours. From there supply will be allocated based on the demand the node is seeing.

Removing a node is the same in the case of a node failure. When we remove a node from the cluster the *best buddy node* will inherit its supply.

4. CONCLUSION

We have created a working prototype for this algorithm and DRL with performance indicators has been proven to achieve convergence in GDRL[3]. However, there is still much room for improvement. One of the main research areas of DRL algorithms is figuring out the fastest way to achieve convergence. Given more time we would of liked to provide benchmark data on how fast our algorithm converges. Other areas we would like to explore is mimicking node failures in our current prototype. The current prototype only supports adding and removing nodes but cannot mimick netsplits and node failures.

5. REFERENCES

[1] Panos Ipeirotis, *The Google attack: How I attacked myself using Google Spreadsheets and I ramped up a* $1000 bandwidth bill$. http://www.behind-the-enemy-lines.com/2012/04/google-attack-how-i-self-attacked.html

[2] Barath Raghavan, *Cloud Control with Distributed Rate Limiting.* http://www.cs.ucla.edu/classes/cs217/SIG07Award.pdf

[3] Rade Stanojevic, *Generalized Distributed Rate Limiting.* http://www.hamilton.ie/person/rade/IWQoS2009.pdf

[4] Hussam Abu-Libdeh, *Ajil: Distributed Rate-limiting for Multicast Networks.* http://www.cs.cornell.edu/projects/quicksilver/public$_{p}df s/ajil.pd$