Distributed Policing with Full Utilization and Rate Guarantees

by

Albert C. B. Choi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A network service provider typically sells service at a fixed traffic rate to customers. This rate is enforced by allowing or dropping packets that pass through, in a process called policing. Distributed policing is a version of the problem where a number of policers must limit their combined traffic allowance to the specified rate. The policers must coordinate their behaviour such that customers are fully allowed the rate they pay for, without receiving too much more, while maintaining some semblance of fairness between packets arriving at one policer versus another.

A review of prior solutions shows that most use predictions or estimations to heuristically allocate rates, and thus cannot provide any error bounds or guarantees on the achieved rate under all scenarios. Other solutions may suffer from starvation or unfairness under certain traffic demand patterns.

We present a new global "leaky bucket" approach that provably prevents starvation, guarantees full utilization, and provides a simple upper bound on the rate allowed under any incoming traffic pattern. We find that the algorithm guarantees a minimum $1/n$ share of the rate for each policer, and achieves close to max-min fairness in many, but not all cases. We also suggest some experimental modifications that could improve the fairness in practice.

Acknowledgements

I am very grateful to my supervisor, Ian Munro, for the many discussions and suggestions on the topic, as well as improving the clarity and readability of this thesis. I would like to thank the people at Cisco Systems for introducing me to this topic, especially Sharanya Subramanian, Herman Levenson, Ketan Padwekar, Allan Lue, and Sandy Wang. I also thank Pat Nicholson for his interest and important insight into the fairness analysis. Finally, I would like to thank Martin Karsten and Gordon Agnew for taking the time to read my thesis and provide feedback and suggestions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Networks are heavily used by large numbers of users, but network service providers have finite capacity for traffic. While there has been prior research on the problem of dynamically dividing the capacity between variable numbers of users and pricing by usage [12, 11, 22], this business model is still not widely used and not always desirable. Users often prefer a predictable level of service and predictable costs. To that end, providers typically sell a fixed traffic rate to customers at a flat cost per month. Thus, even though providers may have capacity to spare, they are limiting the maximum rate that the customer may use. This process is known as rate-limiting, rate-enforcing, or *policing*.

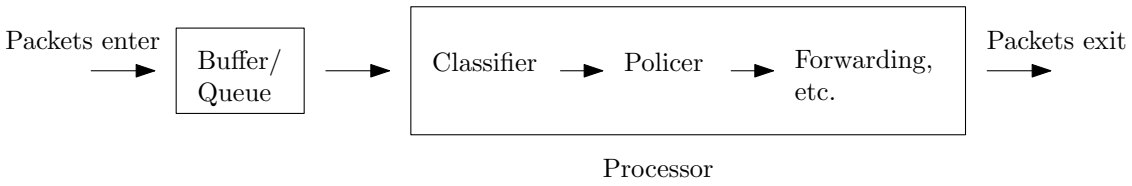
In policing, an algorithm is applied to each incoming packet to decide if it falls within the rate limit or should be dropped. Distributed policing comes into play when it is necessary to enforce an aggregate rate using multiple network processors that each receive a distinct subset of the packets. There are at least two applications for this problem.

One application suggested by Raghavan et al. [17] is for controlling "cloud-based" distributed services. These are computing services running on servers throughout the Internet. A customer accesses the services remotely and does not care which server is used, and their traffic may be distributed onto more than one server. The argument is made that customers prefer to pay for an agreed upon rate instead of usage-based pricing, and as such, introduces the distributed rate limiting (DRL) problem, which we refer to as *distributed policing*.

Another application is inside network hardware, such as a router or switch. Network administrators may want to enforce a rate limit on an arbitrary class of traffic going through the switch, or an Internet Service Provider may sell service at a fixed rate to customers sharing the switch. As network speeds continue to rise, it will become more common for network hardware to have multiple processors, with each one handling policing on a different portion of traffic. Thus we need to scale the policing algorithm to multiple processors.

The service rate sold to a customer in this way is a contract. If customers pay for a flat rate of 5 Mbps, they expect to be able to use any amount of bandwidth up

Figure 1.1: Diagram of a Network Device



to that speed before being artificially rate-limited. It is important that a policing system does not start dropping customer packets before the rate limit is reached. If we over-restrict at first and try to compensate with higher rates later, but higher demand does not occur later, then not only has the customer been unable to use the full rate, but we have also failed to honour the contract.

The focus of this thesis is to provide new distributed policing algorithms and theoretical analyses that guarantee these rates for the customer.[1] We find that previous solutions to the problem lack these guarantees and error bounds, and have only been analyzed in cases where users are well-behaved and follow the TCP congestion control protocol. We are able to provide upper bounds and lower bounds on the global rate under any traffic conditions. However, we fall short of our fairness goal of providing max-min fair rates to each individual policer, and only provide a loose lower bound of $1/n$ rate fraction for each individual.

The rest of this chapter defines the distributed policing problem and the goals we are aiming for in detail. It also provides background on centralized policing algorithms. Chapter 2 describes previous solutions to the distributed problem and provides a detailed comparison of their strategies and shortcomings. Chapter 3 explains our new algorithms and analyses their theoretical worst case performance. We also have some simulation results, which we list in Chapter 4 to confirm the theoretical results and explore further issues.

1.1 Background and Model

1.1.1 Basic Network Device

The model for the centralized policing is a basic network device such as a switch or router (see Figure 1.1). Data packets arrive at the device and are either passed on, or dropped. A packet must be passed on (*i.e.* not dropped) for the customer to have received service. In the device, there is a single processor that handles one packet at a time, examines it, and decides what is to be done with the packet (*e.g.* where to forward it). It is this processor that implements policing, or rate-limiting.

The first step is packet classification. A variety of packets may be arriving on the network device, and the classifier examines a packet's headers to assign it to

[1] By guarantee, we mean that we guarantee not to limit traffic below the agreed upon rate bound. We do not guarantee packets will actually make it to their destinations.

a predefined traffic class. (Generally the point of policing is to ensure Quality of Service for all other traffic that shares the device, so there is little point in policing without more than one traffic class.) If this traffic class has a rate limit assigned, then the policer will decide whether to allow or deny the packet. Finally, if the packet is not dropped, it will go on with other processing and exit the device.

Note that because the processor can only handle one packet at a time, there is a memory buffer for storing incoming packets until the processor is ready for them. If the buffer becomes full, packets may be dropped before the classifier and policer have a chance to look at them. The problem of proper buffer size, number of queues in the buffer, etc. is another research area in itself and is beyond the scope of this project. We assume the buffer is sufficient. We will define traffic demand as the packets that enter the network processor after buffering, so our analysis is independent of buffering effects.

1.1.2 Policing Definition

Policing is applied on a packet-by-packet basis. The main component of the policer is called the metering algorithm, or meter, because it measures the rate of traffic going by. The metering algorithm decides whether the incoming packet is conforming to or exceeding the desired rate. Depending on whether the packet "conforms" or "exceeds," the policer will perform a different action. Most often, conforming packets will be permitted, while exceeding packets will be dropped. Other actions could be to let the packet through marked with a lower priority, or placing into a queue. The action actually taken can be prescribed by a network administrator and is irrelevant from the perspective of the metering algorithm. The metering algorithm will only count packets that conform. In other words, the precise job of this algorithm is to enforce a limit on the number of packets that it will mark "conform."

However, if packets that exceed are queued, it is no longer considered policing, but traffic shaping. Suppose three packets arrive in the order A, B, C when the meter reports the rate is exceeded, and a fourth packet D arrives when enough time has passed that the rate is under again. In policing, packets A, B, C are simply dropped, and D would be allowed. In traffic shaping, A, B, C are placed in a queue. Once enough time has passed, the first packet to be allowed will be A. When D arrives, it would be placed in the queue behind B, C. Shaping adds additional complexity. Not only does the meter have to signal the queue, the output of the queue needs to lead back to the metering algorithm, so previously exceeded packets can be counted again as conformant. Thus, even though the same metering algorithm can be used for policing and shaping, the data flow of traffic shaping does not fit our policing model.

In a policer, the action taken does not affect the meter, so we will use the term *policing algorithm* to refer to the metering algorithm. In this document we will also interchangeably use the terms *permit*, *allow*, or *forward* a packet when we mean to

mark it "conform" and the terms *drop* or *deny* a packet when we mean to mark it "exceed." We use the terms policing to rate r or limiting the rate to r to mean that the rate at which packets are marked "conform" is limited to r.

The metering procedure described above is sometimes called a two-colour marker, since it categorizes packets into two colours: conform and exceed. There also exist three-colour policers, such as the single rate [6] and two rate [7] three-colour markers. We will only consider the two-colour markers.

1.1.3 Distributed Model

In the distributed model, there are several policers, but still only one policer per processor. In both the multi-processor router and the cloud service applications, each packet only needs to be serviced by one processor or server. Therefore each packet is counted by only one policer. In the general rate control problem, a network is described as a graph of links, and packets may flow through more than one link to reach their destination. Under this terminology, our model is a set of independent links. We will simply call it a set of policers.

1.2 The Distributed Policing Problem

We define n policers or nodes $P_1, \ldots, P_n$. Each policer receives a different portion of incoming traffic, which we call the demand, at rates $d_1, \ldots, d_n$. These rates may be fluctuating. Each policer must locally run an algorithm to allow or deny each packet that it receives, such that the total rate allowed by the set of policers is no more than r and no less than $\min(d, r)$ where $d = \sum d_i$ is the total demand received. Allowing more than this is measured as error in correctness, and allowing less is considered inability to fully utilize the rate. It is permissible to have small error in either direction, but from a customer satisfaction standpoint, error in the correctness is preferred.

We assume each policer has capacity greater than r, so that any single policer can support the full rate r. That is, there are no extra restrictions on which distributions of resources are possible in the system. However, some distributions are better (more fair) than others; the ideal distribution is that of max-min fairness, which we elaborate on in Section 1.2.2. To achieve these requirements, each policer is able to communicate with any other policer as needed. Of course, less communication is preferred.

1.2.1 Ideal Qualities

The following list summarizes the qualities of a "good" solution, roughly in order of importance.

1. *Correctness.* The algorithm for policing is correct if it limits the combined traffic to the desired rate r. Allowing traffic higher than r would be error in the correctness. This error is often defined in terms of the burst size B, which means in a time interval of length Δt, the allowed volume may be up to $r\Delta t + B$. Then in an arbitrarily small timeframe, a burst of traffic size B could be allowed.

2. *Full Utilization.* The algorithm should use all available resources, *i.e.* it should not over-police. Forcing traffic to be at a rate slower than $\min(d, r)$ would be over-policing, as packets have been unnecessarily dropped. If full utilization were not a key issue, then the simplest solution would be to allocate $1/n^{th}$ of the rate to each policer.

3. *Fairness/Starvation.* Resources should be divided fairly. Fairness can be interpreted several ways, which we will describe in the next section. The minimal notion of fairness is that each policer should permit some (non-zero) level of traffic, *i.e.* avoid starvation.

4. *Minimizing Overhead.* Each policer is independent with no shared memory between them. There must be some communication between policers to correctly achieve the aggregate rate limit. Communication is in the form of network packets, which constitute overhead in the system. Overhead should be minimized.

5. *Robustness to Communication Latency and Loss.* The communication packets will take time to travel between policers. This latency may be higher in a distributed server setup versus a multi-processor switch. If a policer is waiting for a communication packet, it should be able to continue operating normally until it arrives. If the communication packet never arrives, *i.e.* there is communication packet loss, the distributed algorithm should not be adversely affected. Ideally, it should be able to recover.

6. *Minimizing Burstiness.* We would like for the rate of the output traffic to not swing up and down to extreme levels. That is, the rate should be steady, not bursty, if possible.

In this paper, we present new algorithms that first achieve correctness and full utilization, and approximate fairness. We also touch on the other points.

1.2.2 Fairness

Past work has looked at solving distributed policing with different fairness goals in mind. Both Raghavan *et al.* [17] and Stanojević and Shorten [19] suggest the distributed algorithm should behave the same as a centralized algorithm. However, there are at least two centralized algorithms with different fairness properties. Here we justify our choice of max-min fairness.

Max-min fairness (MMF) is a scheme that can be calculated as follows. First give equal shares to each participant. If a participant does not use all of its share, the excess is split amongst the remaining participants evenly, and this is repeated until all resources are used. Mathematically, there is a maximum share v such that all particpants requesting less than v receive their request. All requesting more than this receive v. The value v is the one that fully uses the resource.

Max-min fairness is most equal and it does not encourage any participant to artificially inflate their demand. For example, if a user wanted 4 Mbps rate but the MMF scheme only allows it 2 Mbps, then the user can not possibly increase its share by increasing its request rate. Congestion control protocols such as TCP window [8] ask that users decrease their send rate when the system can not support their rate. With an incentive-compatible distribution like MMF, there is no incentive for a user to violate such congestion control protocols, so there is a good chance of users complying to avoid congestion.

Kelly [12] proposed *proportional fairness* as a utility-maximizing scheme, defined as follows. A vector of rates $x = \{x_s, s \in S\}$ is proportionally fair if it is feasible and if for any other feasible vector x^*, the aggregate of proportional changes is zero or negative:

$$\sum_{s \in S} \frac{x_s^* - x_s}{x_s} \leq 0 \tag{1.1}$$

In some situations, a large increase can be given to one individual while only removing a small amount from another. In these cases, proportional fairness gives optimal utility where max-min fairness does not [1, 12]. In our problem, this is not possible, so the MMF allocation is equivalent to the proportional fair one. Here, utility is maximized simply when total rate is maximized, *i.e.* full utilization.

We define *demand-proportional fairness* as giving shares that are proportional to the demand. This way, each participant gets the same proportion of resource that it asks for. However, it is not incentive-compatible. Still, demand-proportional fairness is quite common, as emulating a single best-effort policer naturally leads to it.

Lastly, the commonly used Jain's fairness index [9] evaluates the equality between rates, which is only required for a subset of max-min fair cases, and almost never for demand-proportional.

Flows vs Policers

The concept of fairness can be applied to any problem where we need to divide a resource amongst particpants. In our problem, we can either define the participants as the policers or flows. A flow is defined as the subset of traffic with the same 5-tuple: source address, destination address, source port, destination port, protocol. Flows roughly correspond to users of a network. Each policer may have a different number of flows, and the numbers may vary widely as flows can start and stop,

often with short lifespans. Ensuring fair allocation between flows is an ideal goal, but more complex than between policers, given their dynamic nature.

There have been prior solutions to distributed policing that attempt to achieve fairness between flows by Raghavan *et al.* [17] and Stanojević and Shorten [19]. However, the algorithms have always been studied with a constant number of long-lived flows, which may not reflect dynamic conditions in the real world. This is understandable, but the algorithms take a noticeable delay to adjust to a change in flow numbers, which would be problematic in case of very short-lived and erratically changing flows. Rapidly changing conditions also make it difficult to accurately measure and evaluate the rates. We think it is prudent to start with fairness between policers, where the traffic levels should be less variable.

Rate as a Resource

Using rate in a resource sharing problem raises additional questions. What is the time interval over which we should compute the rate? If a participant has been idle for several hours, then starts to generate lots of traffic, should it be given credit for the previous idle time? If so, how far back should we look? We do not know the answer to this question. We note that policing algorithms with a constant burst size naturally limit such a credit; no matter how long traffic has been idle, there is a limited burst of traffic that can occur before it is considered at the rate.

In this thesis, we always measure the allowed rate compared to demand over the same time interval. Since packets exiting the policer must be a subset of packets entering it, it is also simpler to measure the rate over a certain volume of traffic. For theorems that we prove, the rate is an average based on the amount of time it takes for a certain volume of traffic to be allowed. In experiments we only examine fairness under steady rate conditions.

Which Fairness Criteria to Aim For?

We focus on max-min fairness between policers, which is simpler than achieving it between flows. However, our methods can be treat flows as participants as well, given a way to identify flows and sufficient space requirement. In comparison to previous solutions, we note that only Flow Proportional Share (FPS) [17] and Distributed Deficit Round Robin (D2R2) [19] explicitly try to achieve max-min fairness between flows. Both C3P and GRD try to emulate a centralized best effort limiter, which gives demand-proportional fairness because every packet has the same probability of being forwarded. Curiously, these two methods were evaluated on giving equal shares to flows, but demand-proportional fairness would only give equal shares if demand for each flow is equal!

1.3 Centralized Policing Algorithms

Before trying to solve the distributed policing problem, it is essential to understand the centralized (*i.e.* non-distributed) policing solutions. Virtually all distributed policing solutions also employ one of the centralized algorithms as a fundamental part. The centralized policing problem refers to limiting the rate of packets going through a single policer. Solutions include the well known *token bucket* [16, 20] and *leaky bucket* [21, 3] algorithms. Sometimes a *virtual queue* [5] algorithm is used. While the implementations differ, all three of these solutions are functionally equivalent to each other. In this section, we describe the leaky bucket algorithm and demonstrate how the others are equivalent.

1.3.1 Leaky Bucket

Conceptually, we have a bucket with a hole in the bottom, so that it leaks out at a fixed rate. One can add to the bucket as long as doing so does not cause it to overflow. The algorithm is characterized by two constants: the bucket depth B (also called burst size) and leak rate r. The level of the bucket is a variable b, initially 0. The value of b is steadily decreasing at the leak rate, but may not decrease below 0. Every time a packet arrives, we check if there is enough space in the bucket for the whole packet. If there is enough room, then the packet size is added to the bucket level, and the packet conforms (is allowed). Otherwise, the packet is dropped, and the level is not increased. This decision is outlined in Algorithm 1.1. It is not hard to see that the rate of packets being allowed is limited by the leak rate from the bucket.

Algorithm 1.1 Leaky Bucket Algorithm

On event: Receiving packet size s

```
if b + s ≤ B then
    Allow packet
    b ← b + s
else
    Drop packet
end if
```

Avoiding packet size bias. This version of the leaky bucket gives a bias against large size packets when the bucket is close to full [2]. Since there is only $B - b$ space available in the bucket, a packet of size $s_1 > B - b$ would be dropped, whereas a smaller packet of size $s_2 < B - b$ would be allowed. In the policing model, dropped packets are not queued, so a large packet being dropped would not block a smaller packet that arrives right after it. If small packets arrive at sufficient rate, there might never be enough space created to allow a large packet, effectively denying all large packets. This bias is avoided using a variant of the leaky bucket algorithm that allows packets as long as $b \leq B$, without regard to the incoming

packet size. The side effect is the bucket can then become "overfilled," *i.e.* $b > B$. For this reason, B is sometimes called the bucket threshold level, rather than the bucket size.

Burst size. After a long enough period of inactivity, the bucket will be empty. Then the algorithm may allow a burst of up to B traffic in very short time. Thus, the bucket size B also gives the burst size. To be more accurate, the burst size is the maximum level that b may reach starting from 0. In the above packet size-agnostic variant, the burst size may actually be $B + max\_packet\_size$.

Finally, when the leaky bucket is used for traffic shaping, it is sometimes called the buffered leaky bucket [23]. Here, a leaky bucket will refer to the unbuffered leaky bucket used for policing.

1.3.2 Token Bucket

The token bucket algorithm [16, 20] is another common method to do (centralized) policing. The token bucket fills with credits, called *tokens*, at a fixed rate, and tokens are removed when packets are allowed. This is exactly opposite to a leaky bucket, but provides equivalent behaviour. The token bucket can be thought of as an inverted leaky bucket; a complete analogy is given in Table 1.1.

Table 1.1: Comparing Token Bucket with Leaky Bucket.

	Token Bucket	*Leaky Bucket*
Concept	Tokens represent credits available to use.	Bucket level represents packets using up the rate.
At regular intervals	Tokens replenish at rate r	Bucket leaks at rate r
	Tokens accumulate until bucket is full	Bucket leaks until it reaches 0
On incoming packet, size s	If enough tokens, subtract s tokens; the packet "conforms." If not enough tokens, the packet "exceeds."	If enough space, add s bytes; the packet "conforms." If not enough space, the packet "exceeds."
Bucket size, in relation to burst size	If bucket is full of tokens, then a burst of traffic could be allowed all at once.	If the bucket is completely empty, then a burst of traffic could be allowed all at once.

1.3.3 Virtual Queue

The concept of a leaky bucket filling with arriving packets and leaking at rate r can also be thought of as a virtual queue filling with packets and being serviced at rate

r. Gibbens and Kelly [5] first proposed using such a virtual queue to mark packets, and Stanojević and Shorten [19] use virtual queues to do basic policing in their distributed C3P and D2R2 algorithms. Unfortunately, implementation details are vague; the description given in [19] is very brief:

> On each arrival the packet size is placed in [a] virtual queue. If the packet is discarded from the virtual queue, then the arriving packet is dropped, otherwise it is forwarded to the appropriate output line. Thus, no queueing delay is caused by any limiter.

The leaky bucket and virtual queue are equivalent assuming items can be partially removed from the queue, *i.e.* byte by byte, otherwise the removal of large items from the virtual queue would be delayed. Since the queue is only virtual, and do not contain any actual items, this assumption is likely valid. The queue model is often useful for analysis. For example, Butto *et al.* [3] modelled the leaky bucket as a G/D/1/B queue (general distribution of arrivals, deterministic service rate by 1 server, and finite waiting room B) in order to analyze leaky bucket's efficacy at policing a bursty source.

Chapter 2

Literature Review

2.1 Padwekar's Algorithm

Techniques for distributed policing are new, indeed the earliest algorithm we know of dates from a US patent entitled "System and Method for Distributed Policing" filed in 2005 by Padwekar [15], and granted in 2006. The algorithm is intended for application inside a network device that has multiple forwarding engines and policers. It is expected to be implemented in some Cisco enterprise-level routers. This algorithm treats the problem as a synchronization problem. The main idea is to have each policer run a copy of a leaky bucket algorithm that polices at rate r, and synchronize the copies via update packets. Since a leaky bucket algorithm only really needs to track one variable — the level of the bucket — this amounts to synchronizing multiple copies of a single variable.

Each policer P_i keeps a copy of the bucket level b_i that leaks at rate r and tracks global usage, as well as a local count LC_i that tracks only local usage. The bucket b_i is called the global bucket because it includes information about all packets permitted globally, though that particular copy is stored locally. Each time a packet is permitted by the policer, LC_i is incremented by the size of the packet. When an update is triggered, a policer broadcasts its local count to all other policers, and the value is merged with their global buckets, hence keeping the global buckets synchronized at every update.

The policing decision is basically the same as for the regular leaky bucket, but taking into account both global and local usage. If the sum of the two is above a certain threshold B, packets are dropped. A copy of the original code filed from the patent is below, but we give a simplified and equivalent version in Algorithm 2.1 using our notation.

```
if (global bucket + local traffic count - (leak rate * elapsed time)
        > burst)
    police packet();
```

```
else
    permit packet();
    local traffic count += packet length;
```

Algorithm 2.1 Policing decision

On Event: Policer i receiving regular packet:
  if $b_i + LC_i > B$ **then**
    Drop packet
  else
    Allow packet
    $LC_i \leftarrow LC_i +$ packet length
    Check condition for sending update
  end if

Updates are simply triggered by local usage, *i.e.* when the local count exceeds some local threshold LT. This is shown in Algorithm 2.2.

Algorithm 2.2 Condition for Sending Update

  if $LC_i > LT$ **then**
    Send update packet(LC_i) to synchronizer
    $b_i \leftarrow b_i + LC_i$
    $LC_i \leftarrow 0$
  end if

Although we described this algorithm as a peer-to-peer system, Padwekar's implementation distinguishes one policer as a synchronizer (master) and the other $n-1$ as non-synchronizer (slave) policers. "Local updates" containing local usage count are sent to the synchronizer, and the synchronizer broadcasts "global updates" containing the new global bucket level to everyone else. This system, shown in Algorithms 2.2–2.3, helps keep the system synchronized even if updates are lost.

Algorithm 2.3 Update Handling

On Event: Synchronizer policer i receiving update packet(LC_p):
  $b_i \leftarrow b_i + LC_p$
  Send update packet(b_i) to non-synchronizers

On Event: Non-synchronizer policer j receiving update packet(b_p):
  $b_j \leftarrow b_p$
  Check condition for sending secondary update

2.1.1 The Deadlock Problem and Secondary Updates

In Padwekar's filing [15], it was noted that there were some deadlock cases "due to clock skew and traffic patterns". The term "deadlock" is actually referring

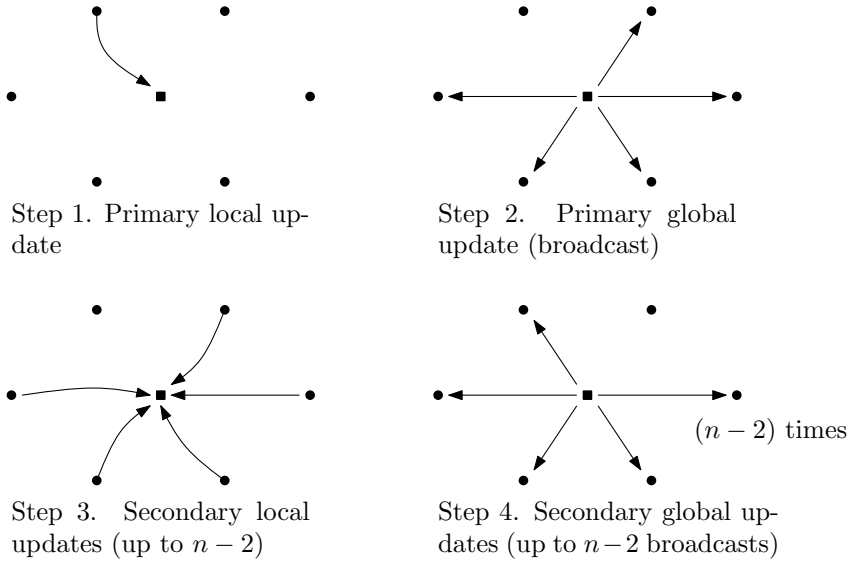
to starvation due to bias against a policer. This happens because the algorithm described above synchronizes the global bucket b_i variable, but not the local counts LC_i. Since the policing decision considers the sum $b_i + LC_i$, this method biases against high LC_i values, which in turn biases in favour of policers that update earlier. The first policer to update has its local count reset to zero, and the update forces other policers to stop allowing traffic, leaving them stuck at a high LC_i value, unable to send their own updates. This is biased against all but the earliest updating policer, and easily causes starvation.

The solution to the deadlock problem proposed by Padwekar [15] is to add a secondary update system. Effectively, the first (primary) update triggers all other policers to send a secondary update as well, allowing all policers to synchronize $LC_i = 0$. Each is followed by a broadcast from the synchronizer, so the process involves 4 steps, as depicted in Figure 2.1. To reduce overhead, there are a few conditions where a secondary update is not necessary to be sent. The specific trigger condition is shown in Algorithm 2.4.

Algorithm 2.4 Condition for Sending Secondary Update

if $LC_i > 0$ **and** $b_i + LC_i > B$ **then**
  Send secondary update packet(LC_i) to synchronizer
  $b_i \leftarrow b_i + LC_i$
  $LC_i \leftarrow 0$
end if

Figure 2.1: The update process in Padwekar's Algorithm takes 4 steps.



2.1.2 Issues with Padwekar's Algorithm

After the discovery of deadlock or starvation case, and the secondary update system to solve it, there was a question of whether this was enough to prevent starvation entirely. In this thesis, we show that it cannot. This motivates our search for an algorithm that provably prevents starvation, which we present in the next chapter.

Starvation Case

Padwekar's full solution periodically synchronizes each policer to the same bucket value. However, each policer makes no assumptions about the other policers' usage in between updates, and each one may be permitting traffic at the full rate r by itself. While this ensures full utilization, it often leads to overusage. When the synchronization occurs, the bucket value will be much higher than the threshold B, and the leaky bucket algorithm will drop all packets until the overusage is compensated for.

The problem is this compensation or "drop" period, which always occurs when there is more than one policer receiving total demand greater than the limit. It is exacerbated by the synchronization procedure which ensures all policers enter (and leave) the drop period at the same time. Since each individual policer is allowed to use the total rate limit r during an "allow" period, the total usage rate during such a period could be nr. To compensate then, the "drop" period would last $n-1$ times longer than the "allow" period! A starvation case would occur if one of these policers happens to only receive bursts of traffic during the drop periods, because it would *never* permit a single packet. The problem arises because the timing of the drop period is imposed on a policer by external sources. In our new algorithms, we solve this by allowing a policer to have more control over when it may permit traffic.

Burstiness

From the example above, we see another problem – the output traffic is quite bursty. Since all policers can allow traffic at rate r, we get alternating periods of high traffic (up to nr) and no traffic.

Overhead

The system broadcasts its updates, which is high overhead relative to other algorithms, especially under the secondary update system where single update triggers not only one broadcast, but up to n broadcasts. This is $O(n^2)$ update packets. More precisely it is $n-1$ primary updates and up to $(n-2)(n-1)$ secondary updates, for $(n-1)^2$ in total. This is not as bad as it sounds. Without secondary updates, a broadcast is triggered for every LT bytes allowed by each policer. If a

secondary update is triggered, say on policer j, then a broadcast has been triggered after j allowed LC_j bytes instead of LT bytes, where $LC_j < LT$. If all the values of LC average half of LT, then the secondary update system produces double the number of update packets as it does without secondary updates.

The system produces $n-1$ update packets for every LC_{avg} bytes allowed by the whole system, where LC_{avg} is the average of all non-zero local count values. Finally, due to the 4 distinct steps of the update process (Figure 2.1), the synchronization will take 4 times the communication latency to complete. Overall, in terms of overhead, this broadcast method is likely to be the least efficient of all the methods reviewed.

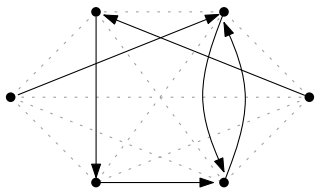
2.2 Techniques of Raghavan *et al.*

In 2007, Raghavan *et al.* [17] proposed three distributed policing algorithms, to be applied to rate control of cloud-computing services over the Internet. Unlike Padwekar, these algorithms do not try to synchronize a variable. Rather, they allocate different rate limits to each policer, based on expected demands at each policer. The main communication between policers is simply to update the new demand estimates. However, since it is impossible to predict exact demand, a reliable update system was not a priority for the authors. The method is inherently error-prone and only gives an approximate solution, but is fully distributed and requires much less communication overhead than Padwekar's system. The algorithms were evaluated based on a network simulator.

2.2.1 Gossip Protocol for Communication

All of Raghavan *et al.*'s algorithms use the same "gossip protocol" to send updated demand estimates. The protocol is less expensive than a full broadcast. At fixed time intervals, each policer i sends its demand estimate $\tilde{d}_i$ to a fixed number of randomly chosen policers, as depicted in Figure 2.2. The recipients may be selected from any of the policers. If policer j does not receive an update from policer i, it simply continues to use the last known estimate for i.

Figure 2.2: Gossip Protocol communication graph with one randomly chosen recipient per node.



2.2.2 Global Token Bucket (GTB)

Global Token Bucket (GTB) [17] is a naïve algorithm presented mainly for comparison and expectedly does not perform well. Each policer employs a token bucket algorithm that refills at rate r, but also removes tokens based on expected (predicted) demand at other policers. On the surface, GTB looks like Padwekar's algorithm, but it is not. Since predictions are unlikely to match real demand at other policers, token buckets quickly go out of sync, and these errors are *never* corrected. This differs from Padwekar's, where all buckets are synchronized.

We find that a better characterization of GTB is that every policer P_i uses a token bucket that allows packets at some rate r_i, where

$$r_i = r - \tilde{d}_{\text{outside}} \tag{2.1}$$

and $\tilde{d}_{\text{outside}}$ is simply the sum of demand estimates at all policers other than i:

$$\tilde{d}_{\text{outside}} = \sum_{j \neq i} \tilde{d}_j \tag{2.2}$$

There are obvious problems with this rate allocation. For example, any policer will starve if outside demand is greater than r, as this sets $r_i = 0$.

2.2.3 Flow Proportional Share (FPS)

Flow Proportional Share (FPS) [17] is the algorithm proposed by Raghavan *et al.* to be used for policing TCP (Transmission Control Protocol) flows. The main claim is to provide a fair rate to each TCP flow, even while there are variable numbers of TCP flows at each policer. FPS allocates individual rates r_i to each policer i based on the number of flows, but does not actually count the flows. It uses a heuristic to estimate the number of flows based on demand. Like the GTB algorithm, it uses a token bucket at each policer to regulate the individual rate.

The idea is to allocate a rate to each policer that is directly proportional to the flow count. The reasoning is that flows arriving on the same policer will compete with each other, resulting in roughly equal shares, and with this allocation, all flows globally would get roughly equal shares. There are two assumptions here. First, this relies on the TCP protocol for congestion control [8], namely, each TCP flow should increase its transmission (demand) rate gradually, and then slow down once there is congestion, due to missing TCP acknowledgements from the destination. Otherwise flows could not compete as stated. Raghavan *et al.* only recommend FPS for TCP traffic. Second, equal shares for all flows is not the ideal (max-min fair) allocation if there are some flows that have low demand (called bottlenecked flows by Raghavan *et al.*). To work around this, FPS computes weights, and gives rates to each policer proportional to its weight.

The algorithm for computing the weight is shown in Algorithm 2.5. In the case where local demand $\tilde{d}_i$ exceeds the local rate limit r_i, FPS assumes that all the unbottlenecked flows are already getting roughly the same rate. So FPS samples the rates of some flows and uses the maximum flow rate as an indicator of the rate of each unbottlenecked flow. For example, if the local rate limit was 50 Kbps, there is one slow (bottlenecked) flow using 10 Kbps, and two unbottlenecked flows getting 20 Kbps, then the weight would simply be $50/20 = 2.5$ "adjusted flows". In the case where $\tilde{d}_i < r_i$, all flows are slow (bottlenecked), so FPS sets the weight so that the resulting new rate limit r_i equals $\tilde{d}_i$.

Algorithm 2.5 FPS-Estimate()

if $\tilde{d}_i \geq r_i$ **then**
  maxflowrate $\leftarrow$ MaxRate(sample set)
  weight $\leftarrow r_i$ / maxflowrate
else
  remoteweights $\leftarrow \sum_{j \neq i}^{n} w_j$
  weight $\leftarrow \frac{\tilde{d}_i \times \text{remoteweights}}{r - \tilde{d}_i}$
end if
$r_i \leftarrow \frac{\text{weight} \times r}{\text{remoteweights} + \text{weight}}$
Propagate(weight)

Issues with FPS

FPS uses a lot of approximations and heuristics. Like GTB, the actual rate achieved may be off from the desired rate, and this error is never compensated for. The assumption that flows at a single policer would automatically compete and share the policer's rate in a fair way requires that all flows be well-behaved. It was later shown by Stanojević and Shorten [19] that the weight used by FPS can be a very poor estimate of actual flow count if the flows have different round-trip times, *i.e.* when some flows accelerate quicker than others.

2.2.4 Global Random Drop (GRD)

Global Random Drop (GRD) [17] is unique in that is the only distributed policing algorithm reviewed that does not use a token bucket variant to do basic rate limiting. Instead packets are randomly dropped, at a probability that gives an expected rate. The probability is determined by the demand estimate:

$$p_{drop} = \frac{\tilde{d} - r}{\tilde{d}}$$
$$p_{allow} = 1 - p_{drop} = \frac{r}{\tilde{d}} \tag{2.3}$$

Then the expected rate allowed is:

$$\begin{aligned} E(a) &= p_{allow} d \\ &= \frac{r}{\tilde{d}} d \end{aligned} \tag{2.4}$$

And if we assume actual demand matches the demand estimate, $d = \tilde{d}$,

$$\begin{aligned} E(a) &= \frac{r}{d} d \\ &= r \end{aligned} \tag{2.5}$$

The random drop method can be used in a centralized policing algorithm, which Raghavan *et al.* [17] call Centralized Random Drop (CRD). In the distributed version, GRD only needs to compute $\tilde{d}$ as the sum of all demand estimates across policers. The drop probability should be the same for all policers (ignoring differences in estimates caused by the gossip protocol), so GRD does not give individual rates to each policer. This makes errors in individual policer demand estimates less important – as long as the overall demand d is close to the overall estimated demand $\tilde{d}$, GRD should be close to the overall rate limit r.

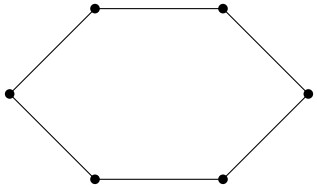
2.3 Stanojević and Shorten's Algorithms

In 2008, Stanojević and Shorten [19] proposed two algorithms – Cloud Control with Constant Probabilities (C3P) and Distributed Deficit Round Robin (D2R2). Unlike Raghavan *et al.*'s methods, these algorithms do not depend on randomness and approximations. They use a feedback mechanism that adjusts rates at each policer until certain performance indicators are equalized, and they are proven to converge.

2.3.1 Communication

For these algorithms, there is an undirected D-regular communication graph between policers. That is, every policer only communicates (*i.e.* sends updates to) its D neighbours in the graph. In their experiments, the degree used was $D = 2$ in a 10 policer graph. Like Raghavan *et al.*'s gossip protocol, this is a great savings over a full broadcast from every policer. The updates are sent at fixed time intervals, on the order of several seconds (2 sec in experiments). This interval is relatively long, *e.g.* compared to Raghavan *et al.*'s update intervals, in order to get good estimation for loss rates.

Figure 2.3: The 2-regular communication graph in C3P and D2R2 is a ring.



2.3.2 Cloud Control with Constant Probabilities (C3P)

The algorithm Cloud Control with Constant Probabilities (C3P) attempts to emulate a centralized best effort policer. In such a centralized policer, every packet has the same probability of being dropped. This is the same goal as Global Random Drop (GRD). However in C3P, no probabilistic drop is employed. C3P allocates rate limits to individual policers and continually adjusts them, until the loss rates at all policers is equal. This is essentially a feedback mechanism. C3P relies on a relation between loss rate and send rate, which is observed in, but not limited to, TCP flows. Additionally, it is shown that the algorithm behaves well for TCP flows. To regulate the rate r_i at a single policer i, a virtual queue is used, which is much like a leaky bucket.

Algorithm 2.6 shows how C3P sets each policer's rate limit r_i, where E are the edges of the communication graph, p_i is the loss rate at policer i, and η is a parameter controlling how quickly the rates change. The paper shows using induction that, under the communication graph update system, the individual rate limits always add up to the total rate limit, *i.e.*

$$r = \sum_{i=1}^{n} r_i \tag{2.6}$$

Algorithm 2.6 C3P Rate Allocation.

InitializeCapacities()
  for $i = 1 : n$ **do**
    $r_i \leftarrow \frac{r}{n}$
  end for
UpdateCapacities()
  for $i = 1 : n$ **do**
    $r_i \leftarrow r_i + \eta \sum_{(i,j)\in E}(p_i - p_j)$
  end for

Stanojević and Shorten prove that C3P converges to the correct solution (all loss rates equal) given that the loss rate p of a TCP flow is related to its send rate x and round-trip time (RTT) as follows:

$$x(p, RTT) = \frac{\theta}{RTT\sqrt{p}} \tag{2.7}$$

It is also claimed that it will converge to the correct solution in general, if the rate limit r_i is related to the loss rate by a differentiable, convex function $f_i : (0, 1) \rightarrow (0, \infty)$

$$r_i = f_i(p_i) \tag{2.8}$$

2.3.3 Distributed Deficit Round Robin (D2R2)

The Distributed Deficit Round Robin (D2R2) algorithm is similar to C3P in that it also sets and updates the rate limits at each policer. It is different in that it tries to emulate a centralized processor sharing queue, *i.e.* giving fair share to each flow that shares the policer. D2R2 is unique in that it is the only algorithm reviewed to use a virtual DRR (Deficit Round Robin) [18] algorithm to control the rate limit at a policer. Where C3P is using a single virtual queue, D2R2 is using a virtual set of queues – the DRR. The virtual DRR at each policer gives max-min fair rates to each flow traversing it, and the D2R2 feedback mechanism continually adjusts the policer rates r_i so that the resulting flow rates are also MMF globally.

The D2R2 rate allocation scheme is to adjust rates so that max-min fair shares are achieved for each flow. In a MMF allocation, there is some *maximum share* given to participants, v, where every participant with demand d_i gets

$$\min(d_i, v)$$

In other words, this is the cut-off point or cap on the demands of each participant, above which no one can get higher. (The paper [19] calls this the "fair-share," but this term is confusing. It is more accurate to call it the maximum share.)

The maximum share given to a flow at policer i can be calculated as the solution to the equation (from [19]):

$$g_i(v_i) = r_i \tag{2.9}$$

where the function g_i is the sum of rates allocated to the flows in $\mathcal{F}_i$ when each flow s has demand $d_s^{(i)}$:

$$g_i(v_i) = \sum_{s \in \mathcal{F}_i} \min(d_s^{(i)}, v_i) \tag{2.10}$$

If each policer i provides a MMF allocation to its flows $\mathcal{F}_i$, and the maximum share v_i at each policer is equal, then we have max-min fairness across all flows globally. The D2R2 algorithm tries to equalize the maximum shares, v_i, by adjusting a policer's local rate limit r_i based on its neighbours' maximum shares. The pseudocode that does this is shown in Algorithm 2.7.

For D2R2, Stanojević and Shorten prove that this method of updating the rate limits r_i, will result in converging towards a maximum fair share $v^{(\alpha)}$ that is at

Algorithm 2.7 D2R2 Rate Allocation.

InitializeCapacities()

for $i = 1 : n$ **do**

$r_i \leftarrow \frac{r}{n}$

end for

UpdateCapacities()

for $i = 1 : n$ **do**

if $\sum_{s \in \mathcal{F}_i} d_s^{(i)} > r_i$ **then**

$v \leftarrow$ solution to $g_i(v) = r_i$

else

$v \leftarrow \max_{s \in \mathcal{F}_i}(d_s^{(i)})$

end if

$R_i \leftarrow r_i - g_i(v)$

$v_i \leftarrow v + \alpha R_i$

$r_i \leftarrow r_i + \eta \sum_{(i,j) \in E}(v_j - v_i)$

end for

most $1/\alpha$ factor away from the ideal maximum share v^*. More precisely, given some demands, the proofs show:

$$v^{(\alpha)} \leq v^* \tag{2.11}$$

$$0 \leq \frac{v^* - v^{(\alpha)}}{v^*} \leq \frac{n}{M\alpha} \tag{2.12}$$

where $\alpha \geq 1$ and M is the number of flows with demand not less than v^*.

2.4 Predictive vs Non-Predictive Algorithms

We can divide the distributed policing algorithms into two main classes: predictive algorithms that work based on expected traffic usage, and non-predictive algorithms that keep track of exact traffic usage.

1. Predictive algorithms look at past performance and decide how much rate limiting to do based on predicted future behaviour. Generally this is effected by assigning individual rates r_i to each policer P_i, although this is not the case for GRD. Each policer controls its own rate using some local policing algorithm (*e.g.* token bucket). The only information shared is the performance indicators (*e.g.* demand estimates, loss rates) necessary to determine the rate allocation. The benefit is that these methods can be fully distributed, with low communication overhead. The global rate limit r is adhered to assuming the individual rate limits r_i sum to r (except for GRD, where global limit is adhered to when total demand matches the demand estimate). Predictive algorithms include:

- Global Token Bucket (GTB)
- Global Random Drop (GRD)
- Flow-Proportional Share (FPS)
- Cloud Control with Constant Probabilities (C3P)
- Distributed Deficit Round Robin (D2R2)

2. Non-predictive algorithms never predict the demand to be received by each policer. Each policer running the algorithm only acts based on information it knows for sure. This necessarily means that global usage information must be tracked and shared to all policers, usually requiring that a single master exist, or that communication be broadcast. Generally, the focus is on tracking and communicating the global usage instead of adjusting rates at each policer. For example, in Padwekar's algorithm, each policer assumes there is no traffic arriving at other policers, but if/when update messages reveal otherwise, then this is corrected. Non-predictive algorithms include:

   - Padwekar's Algorithm
   - Global, leaky bucket algorithms developed in this thesis

It is a generalization to say that predictive algorithms use rate allocation, and non-predictive ones do not. Indeed, GTB was originally described as a global rate algorithm with demand estimates, and GRD does not set local rate limits, whereas our Early Start Steady algorithm (Section 3.3) does allocate local rates without predicting demand. The important distinction between the two classes is whether the algorithm relies on predictions or not.

2.4.1 Common Problems with Predictive Algorithms

There are many good qualities about predictive algorithms, namely the low communication without need for a master, and ease of implementing fairness. Certainly, Shorten and Stanojević's algorithms are quite successful, assuming we do not require exact performance. However, we make the case for a non-predictive distributed policing algorithm. The main problems that pervade all predictive algorithms are correctness and full utilization. This is the nature of using predictions that can only approximate actual usage, instead of explicitly tracking it.

Error Accumulation

It is easier to see the issue with Raghavan *et al.*'s GTB, GRD, and FPS, because these all directly use demand estimates from prior time intervals to determine future rate allocations (or drop probabilities for GRD). When demands change, estimates become outdated and give rise to inaccurate allocations (drop probabilities). In

constantly changing conditions, the allocations are always playing catch-up. In C3P and D2R2, an indirect feedback mechanism is used. Rates are iteratively adjusted based on past performance indicators, eventually reaching desired performance at equilibrium. Again, if demands suddenly change, the performance will be different than expected, and the algorithms must slowly adjust again. The analyses of FPS [17], C3P, and D2R2 [19] only prove that the rates will eventually converge to the correct rates, assuming the demand settles down. Until this happens, the actual rate will deviate from the desired rate by some error, and none of these algorithms provide a bound on the error.

Additionally, these errors may accumulate. None of the reviewed predictive algorithms have any mechanism to track or compensate for past error. Understandably, they lack this because it would require much more overhead to communicate this information. It is unknown whether the accumulated error is unbounded.

Starvation and Low Utilization Cases

There are some extreme examples where the predictive algorithms do not perform well. We construct these by making the demands very different from what was expected.

Consider a policer P_i that is receiving traffic in short bursts in between periods of no traffic. Other policers are receiving lots of traffic. In the GTB and FPS algorithms, the rate allocated is directly proportional to prior demand, *i.e.* 0 for P_i. When P_i's traffic burst occurs, its packets are dropped. By the time the demand estimation reflects P_i's demand, let its traffic burst already be over. We cycle this behaviour once demand estimation equals zero again. Thus, such a scheme can lead to starvation for an entire policer. If we do guarantee a minimum share to avoid starvation, then the total used rate will be higher than desired. This leads back to the error accumulation problem. To be fair, the methods do use a filter to smooth the demand estimates, so the rate limit will unlikely be exactly zero.

In the feedback algorithms, C3P and D2R2, the rates allocated are not directly proportional to demands, but slowly adjusted based on factors such as loss rates. However, it is always possible for traffic patterns to be different from expected. The individual rates assigned should add up to the global rate in all of GTB, FPS, C3P, and D2R2. Therefore anytime the actual demand for one policer is lower than predicted, we will fail to get full utilization.

2.5 Distributed Algorithm Components

To complete the overview of prior work, we break down the common components that comprise a distributed policing algorithm. Such a comparison reveals simple improvements that could be made, and we point these out. However, these are not the main contributions of the thesis.

2.5.1 Global Rate Control

1. A global token (leaky) bucket that replenishes (leaks) at rate r. Each policer maintains a copy of this bucket, which must be regularly synchronized:
   (a) Peer-to-peer updates containing difference values.
   (b) Synchronize with a master value. This requires that an agreed-upon master policer holds the correct version of the bucket at all times. (Padwekar)
2. Allocate individual rates r_i to each policer i, and ensure the sum $\sum r_i = r$. The individual local rates will have to be updated regularly. (GTB, FPS, C3P, D2R2)
3. Probability of dropping packets to give an expected rate r for expected demand $\tilde{d}$. The probability of allowing the packet is set to $r/\tilde{d}$. The total demand estimate $\tilde{d}$ must be updated regularly. (CRD, GRD)

2.5.2 Local Rate Control

In some algorithms, we would like to achieve a local rate limit r_i for each policer, which is different from the global rate r. The obvious way is to run one of the centralized policing algorithms at the policer: token bucket, leaky bucket, or virtual queue. The local rate control methods described are:

1. A token (leaky) bucket that replenishes (leaks) at r_i
2. A virtual queue with service rate r_i (C3P, D2R2)
3. A token bucket at rate r, but increase the token removal rate by the outside demand, $r - r_i$ (GTB as described)
4. A token bucket at rate r, but lower the token replenishment rate by $r - r_i$ (FPS)

It is simple to see that these are all the same. Proof: In a token bucket, the policing decision is $bucket + replenishment - removal - packet > 0$. The equivalence of token buckets and virtual queues was shown in Section 1.3.

2.5.3 Rate Allocation Strategies

Most of the predictive algorithms allocate individual rate limits to each policer. In each case, the formula for the rate limit is based on different variables, such as demand estimate $\tilde{d}$, loss rate p, or maximum fair share v. We list these, plus a couple of obvious modifications:

1. GTB allocates the simplistic rate of $r_i = r - \tilde{d}_{\text{outside}}$. A policer will starve if outside demand is greater than r, as this sets $r_i = 0$. We can think of some trivial modifications to GTB's rate allocation strategy that would work better.

2. Modification to GTB that prevents starvation. Allocate the rate $r_i = \max(r - \tilde{d}_{\text{outside}}, r/n)$, thus guaranteeing a minimum share for each policer. In this case, the sum of all local rates r_i may be over r.

3. Another modification to GTB: Keep the allocated rate as $r_i = r - \tilde{d}_{\text{outside}}$, but define $\tilde{d}_{\text{outside}} = \sum_{j \neq i}(\min(\tilde{d}_j, r/n))$ thus guaranteeing that the outside demand is no more than $r - r/n$. Again, the sum of all local rates may be over r. An example is the case of 2 high-demand policers and large number of zero-demand policers. It would result in allocating a rate of $r - r/n$ to each policer, for $2r - 2r/n$ total rate, which is too high when $n > 2$. In general, you can show that it is bad when there are k high-demand policers and $n - k$ zero-demand policers, whenever $n > k > 1$. We do not consider this further.

4. Max-min fair modification to GTB: Given that we have the a vector of all the demand estimates, we can simply set r_i proportional to policer i's MMF share, given its local demand $\tilde{d}_i$ relative to $\tilde{d}$. Under this version, the local rates add up to r assuming estimates are consistent across all policers.

5. FPS allocates rates that are proportional to a weight, i.e. $r_i = rw_i/w$. The weight is the number of unbottlenecked flows at each policer.

6. C3P allocates rates and periodically adjusts them based on loss rates at each policer. The rate limit is increased if the policer's loss rate is higher than its neighbouring policers' loss rates. It sets $r_i \leftarrow r_i + \eta \sum_{(i,j) \in E}(p_i - p_j)$ where p_i represents the loss rate experienced at policer i.

7. D2R2 allocates rates and periodically adjusts them based on residual bandwidth at each policer. The rate limit is increased if the policer's maximum fair share is lower than its neighbouring policers' maximum fair shares. It sets $r_i \leftarrow r_i + \eta \sum_{(i,j) \in E}(v_j - v_i)$ where v_i represents the maximum share at policer i.

2.5.4 Effective Rates

While the predictive GRD algorithm and non-predictive Padwekar's algorithm do not allocate rates, it is still interesting to compare the effective rates that result when applying these algorithms. We call this the global allowed or achieved rate, a, and the local achieved rate a_i.

1. GRD gives an expected allow rate for a policer, $E(a_i) = rd_i/\tilde{d}$. Each policer gets a fraction of the rate proportional to its fraction of total estimated demand, which gives demand-proportional fairness.

2. Padwekar's algorithm is demand-proportional up to a point. Each policer gets

$$a_i = r\frac{\min(d_i, r)}{\sum_{j=1}^{n}\min(d_j, r)}$$

The rationale is as follows. Recall that Padwekar's algorithm synchronizes all policers' leaky buckets so that they all drop traffic or allow traffic at the same time. Then the amount of traffic a policer allows is proportional to the amount of demand it recieves during the allow period, except the leaky bucket prevents any rate higher than r. Hence, policer achieved rates are demand-proportional up to r.

For all the other predictive, rate allocation algorithms, the achieved rates will match the rates they aim for (demand-proportional for C3P, max-min fair for FPS and D2R2), assuming traffic demand behaves as expected. For FPS, this means the actual demand and number of flows should match the estimate(s) from the previous time interval(s), and for the feedback algorithms C3P and D2R2, the demands must have stayed constant for long enough that the performance indicators no longer change.

2.5.5 Compensation Scheme

In the non-predictive, full utilization algorithms (*e.g.* Padwekar's), each policer assumes that no one else will have demand in the coming time interval. If that is true, there is no problem, otherwise compensation will be necessary. There is more than one way to handle the compensation step:

1. Drop traffic until all overuse is compensated for. This naturally arises from the token or leaky bucket algorithm. The result is alternating a period of high traffic with a (long) period of no traffic. (Padwekar's)

2. Lower the local rate to a constant $r_i < r/n$, using some local rate control. The compensation is slower, but will eventually complete (e.g. twice as long when $r_i = r/2n$). Afterwards, push local rate back to r. This is a simple way to guarantee a minimum share. It alternates periods of high traffic with low traffic.

3. Do not fully compensate the overuse; instead only compensate partially so as to keep the volume of overuse bounded. This allows the local rate to be set to $r_i \geq r/n$. The r_i varies depending on how much overuse there is, setting $r_i = r$ when there is no overuse.

In our new algorithms, we use option 3 to achieve a minimum rate guarantee for each policer.

2.5.6 Update Intervals

1. Use a timer that causes updates to be sent at equal time intervals. (GTB, GRD, FPS: 0.1–0.5 sec; C3P and D2R2: 2 sec)

2. Use a local count LC_i that tracks how much volume (bytes) is permitted by a policer i during the update interval. Send updates every time LT bytes are permitted by the policer. Care is necessary to prevent starvation. (Padwekar's)

Chapter 3

A New Leaky Bucket Approach

In this section, we propose a new class of algorithms for solving the distributed policing problem. We describe several variations on the idea, some of which have better fairness properties than others.

3.1 General Algorithm

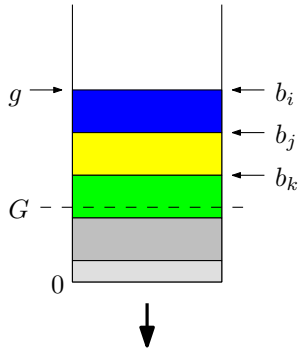
The general approach is to use a global, leaky bucket for controlling the global rate. The global bucket is controlled by a master, which may or may not be located on the same processor as a policer. This bucket has a current level g, which is leaking at a constant rate. Each policer keeps a copy of g, call it b_i, which is occasionally synchronized with the master copy. Unlike previous work, policers are not all synchronized at once; instead we use a *polling* system. By polling, we mean that a policer never receives an update unless it requests one from the master.

Each policer lets through a fixed amount of traffic and reports it to the master. The master adds this amount to the global bucket, and replies (to the reporting policer only) with the new level of the bucket. At this point, the policer's copy b_i is synchronized with the master copy g. It is the new value of b_i that determines when the policer may next report. Since each policer allows a fixed amount of traffic between reports, this also determines the speed at which the policer allows traffic.

A key notion in the methods we introduce is that since each traffic report is *added* to the global bucket level, there is effectively a stack of reports (or stripes) in the bucket. We may imagine each stripe has a colour corresponding to the policer that sent the report, as depicted in Figure 3.1. As the bucket leaks, the entire stack is sinking. The idea is that, after sending a report, *the policer is required to wait for its stripe in the bucket to leak down to some threshold level before sending in another report.* It may report later, but no earlier than this. The policer knows the level of its stripe, because at the time of synchronization, its stripe is the top stripe in the bucket. We stress that this means the master does not need to remember

the positions of all the stripes in the bucket; it needs only to track the level of the highest stripe, which is g, and each policer implicitly tracks the level of its own stripe with b_i.

Figure 3.1: Representation of the global leaky bucket as a stack of stripes sinking at steady rate. Each report from a policer adds a stripe into the top of the bucket. The master remembers the top level g, while the policers i, j, and k remember the top levels of their respective stripes. At most one stripe from each policer is allowed above the threshold G.



This *protocol* only restricts the time and traffic allowed by a policer between reports, but does not specify which packets a policer should allow to meet this requirement. The different versions of the algorithm we present vary in the way policers operate at this level. We use the following notation:

- n: number of policers
- P_i: policer i ($1 \leq i \leq n$)
- g: current level of global bucket
- G: threshold level of global bucket
- b_i: value of g as known to P_i (that is, ignoring any reports from other policers since P_i's last update)
- B: threshold level of P_i's leaky bucket
- r: global rate limit - the rate at which g is leaking
- LT: local threshold - the amount of traffic a policer reports to the master at a time
- LC_i: local count - the amount of traffic P_i has allowed but not yet reported to the master

In the remainder of this section, after briefly discussing related work, we outline key advantages of the general algorithm, namely its low overhead and how it avoids starvation. In Sections 3.2 and 3.3, we give different versions of the algorithm that fully specify each policer's behaviour in order to guarantee full utilization and a good degree of fairness. In Section 3.4, we describe an experimental modification of the algorithm, which sometimes violates the reporting protocol.

3.1.1 Related Work

The model of a bucket with a stack of stripes sinking together means that the stripes are being leaked out in the order in which they arrive. One can also think of this as a virtual queue, which we showed in Section 1.3.3 is equivalent to the leaky bucket. Given that each policer in our algorithm retains the position, b_i, of a different stripe, it is really a distributed virtual queue.

Johnson [10] reviewed several distributed queues, where the storage of a single queue is distributed amongst several processors. One that is similar to our model is the *queue manager* algorithm, where a single queue manager keeps track of which processor is storing what part of the queue. However, this only distributes the storage of elements in the queue, whereas our model also distributes the information about element positions. Johnson's *fully distributed queue* does have this property, but with no communication when processors insert into the queue, it no longer maintains a global ordering. Furthermore, the study of distributed queues emphasizes locating and retrieving elements stored in the queue, but this is irrelevant to our problem, since our virtual queue does not store real elements, only the (virtual) size of elements.

A more closely related technique is the Numbered Ticket algorithm given by Fischer *et al.* [4] for allocating resources in FIFO (first in, first out) order. Consider a line of customers at a bank, with k tellers. If there are fewer than k customers at the bank, then the next customer will be serviced immediately. When there are more, the extra customers must line up and are serviced in FIFO order once a teller becomes available. It is impossible for a single customer to occupy more than one spot in the line. In our model, if a policer enqueues one stripe below the bucket threshold, it is "serviced" right away, and can re-enter the queue immediately. When the bucket level is above the threshold, a policer that enqueues the next stripe must wait before it can re-enter. We enforce this so that a single policer can not occupy more than one spot above the threshold. The Numbered Ticket algorithm solves this FIFO ordering problem by assigning each customer a ticket number, and storing in shared memory the ticket number that may receive service (*valid*) and the next ticket number to be given to a new customer (*issue*). Since the service rate in our leaky bucket is constant, the *valid* number is implicit, but the *issue* number must still be shared; this is g, the top level of the bucket that the master keeps track of.

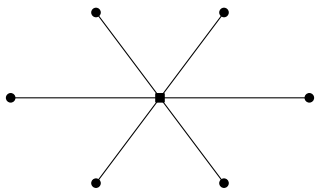
Fairness in the distributed policing problem is more complex than in either of

the above problems however. With the bank problem, it is sufficient to service customers in the same order they enter the queue (FIFO) as this would be fair. In our problem, policers are continuously re-entering the virtual queue/leaky bucket, and we are limiting the rate at which they re-enter, but this *limiting* may not always be fair. For example, one variation could make the policer wait until getting to the front of the queue before it starts allowing traffic, and re-enter the queue when it has allowed enough. It might be more fair to only restrict when the policer may re-enter the queue (*e.g.* after reaching the front of the queue), without restricting when it can accumulate the LT of allowed traffic. These two choices are covered by our Basic and Early Start algorithms. The analogy would be: if customers have to complete some task before entering the bank line, they can either perform the task after reaching the teller, or do it while waiting in line.

3.1.2 Overhead

The overhead is notably less than that of Padwekar's algorithm [15]. As discussed in Section 2.1.2, Padwekar's algorithm requires $n-1$ update messages per LT (or fewer) bytes allowed, whereas our algorithm only needs 2 messages for every LT allowed. In addition, the update completes after 2 times the communication latency (from policer to master and back), rather than 4 in Padwekar's Secondary Update system. The communication graph between the n policers and 1 master is shown in Figure 3.2.

Figure 3.2: The communication graph for the polling algorithm, with the master in the center connected to n policers.



3.1.3 Preventing Starvation

Due to the polling system, any given policer will not know that other policers' reports have occurred until it, itself, has accumulated enough local count to send a report. This makes it impossible for starvation to occur.

Theorem 1. *No policer can ever be starved under the polling system.*

Proof. The protocol restricts a policer's rate by restricting LT traffic to be allowed in between reports. Assume that policer P_i is in a starvation state, *i.e.* it is perpetually rejecting packets. This implies that it has already permitted LT traffic and is unable to send a report. The protocol says that a report may be sent once

b_i leaks below G in the global bucket. We show that b_i can only decrease. In this algorithm, b_i can only increase if P_i itself sends a report, which can not happen in a starvation state. Other policers can not increase its value. Meanwhile, b_i is steadily decreasing at the global bucket leak rate, r. No matter how high the value of b_i, it is guaranteed that eventually P_i will be able to report, after which a new quota of LT traffic is allowed. P_i is no longer in a starvation state, thus a contradiction. □

3.1.4 Properties

Theorem 1 only states that starvation does not occur because a policer is guaranteed to be able to report eventually. We can make this statement even stronger. There is a limit on how long a policer must wait before it can send a report again, because of an upper limit on the values of g and b_i.

Lemma 2. *The amount in the global bucket at any time can be never be more than*

$$g \leq G + nLT \tag{3.1}$$

Proof. No policer can report to the master until the level of their last stripe b_i has sunk below the threshold G. Thus, at most one stripe per policer may be above G, and the most in the global bucket is $G + nLT$. □

Theorem 3. *The protocol by itself never restricts any policer to less than r/n rate of traffic permitted.*

Proof. Consider the time interval between updates for policer i. The amount of traffic permitted in this time is LT. The minimum time interval required by the protocol is when b_i decreases to G, so the maximum rate allowed for P_i during this time interval is:

$$r_i = \frac{rLT}{b_i - G} \tag{3.2}$$

where b_i is the updated value just received from the master as a copy of the global bucket level g. Since Lemma 2 gives the maximum value of g, it can also be applied to b_i:

$$\begin{aligned} r_i &\geq \frac{rLT}{G + nLT - G} \\ &\geq \frac{r}{n} \end{aligned} \tag{3.3}$$

□

We say that the protocol, itself, does not restrict a policer to less than r/n, because other factors may cause the rate to be less than this. For example, the demand may be less than r/n; this is not a problem. More importantly, we find that some versions of the algorithm impose additional restrictions on how quickly a policer *permits* traffic on top of the protocol's restriction of when an update may be sent.

3.2 A Basic Version Achieving Full Utilization

Recall that the b_i value, after synchronizing with the master bucket level, determines when policer P_i is next allowed to report traffic, but does not specify which packets a policer should allow to meet this requirement. Here, we describe one way to do it. The naïve approach is for a policer is to take the b_i value and use it as its own leaky bucket for policing. That is, each policer will leak b_i at the same rate r as the master, and on a packet-by-packet basis, decide whether to allow a packet depending on the level of b_i compared to a threshold B. This version is very similar to Padwekar's algorithm [15], where each policer also runs a leaky bucket, differing only in our polling update system versus a simultaneous synchronization strategy.

The pseudocode for the Basic algorithm is shown in Algorithms 3.1 and 3.2. Particularly, the policing decision requires

$$b_i + LC_i \leq B \tag{3.4}$$

for a packet to be permitted, where LC_i is the amount of traffic allowed but not yet reported to the master. Otherwise a packet is dropped.

Note that the bucket threshold B for locally permitting a packet is different, but related to the global bucket threshold G for when a policer may report a stripe of traffic. The relation is:

$$B = G + LT \tag{3.5}$$

This can be explained as follows. The policer P_i is allowed to report LT traffic when its stripe b_i has leaked down to G at the *earliest*. Also, the policer only reports after its local count LC_i fills up to LT traffic. Given the policing decision in Equation (3.4), this happens at the earliest when $b_i + LT = B$. Since we want $b_i = G$ at the earliest possible report time, then it follows that we should choose the threshold $B = G + LT$.

Effectively this variation only starts permitting traffic once b_i leaks down to $G + LT$. The problem of starting at this time, is that it will not have completed allowing LT traffic by the time b_i leaks down to G, unless the policer's traffic demand is incoming at rate $d_i \geq r$.

Algorithm 3.1 Basic algorithm for master

On event: Master receiving report(LT) from policer P_i

1: $g \leftarrow g + LT$
2: Send update packet(g) to policer P_i

3.2.1 Time-Staggering Property

The Basic algorithm described exhibits time-staggering that greatly reduces burstiness in the global rate compared to Padwekar's algorithm. While individual policers

Algorithm 3.2 Basic algorithm for policer i

On event: Policer receiving update(g) from master

1: $b_i \leftarrow g$

On event: Receiving regular packet

1: **if** $b_i + LC_i \leq B$ **then**
2: Allow packet
3: $LC_i \leftarrow LC_i +$ packet size
4: Check condition for reporting
5: **else**
6: Drop packet
7: **end if**

Condition for reporting:

1: **if** $LC_i \geq LT$ **then**
2: Report(LT) to master
3: $b_i \leftarrow b_i + LT$
4: $LC_i \leftarrow LC_i - LT$
5: **end if**

may still produce bursts of traffic[1], these bursts are staggered and do not coincide with each other thanks to the polling update system and total ordering of reports in the global bucket. A special case is when each policer is receiving traffic demand at r or faster, each policer's share is spread over a different time slice. The result is time-sharing with no explicit scheduling required.

Each time a policer reports, the master bucket is increased by LT. Then, a policer receives back the bucket value that is always LT higher than the one received by the policer before it, minus any leak due to elapsed time. Let P_j be the policer that updates after P_i and assume $i \neq j$. Because all policers leak their b_i value at the same rate, we have the invariant:

$$b_j = b_i + LT \tag{3.6}$$

If b_i is over the threshold B, policer i will have to wait for it to leak down to B before it starts allowing more packets under the Basic algorithm. Let $t_{\text{start}}(P_i)$ represent this start time. P_j will have to wait for LT more bytes to leak than P_i does, so the start times are staggered in the same order as updates:

$$t_{\text{start}}(P_j) = t_{\text{start}}(P_i) + \frac{LT}{r} \tag{3.7}$$

If policer i is receiving demand at rate $d_i \geq r$, then after the start time, it can

[1] We will improve on the individual policer burstiness in Section 3.3.

allow LT more volume in LT/r time. Thus, P_i reports again at time:

$$t_{\text{report}}(P_i) = t_{\text{start}}(P_i) + \frac{LT}{r} \tag{3.8}$$

This is the same time that P_j starts allowing packets. Furthermore, because P_i's new report is sent after P_j, it's new b_i value is necessarily at least LT higher than b_j, and P_i stops allowing traffic.

We can show that the special case of $d_i > r$ for all i leads to a complete time-sharing between policers. If all policers have $d_i \geq r$, then total demand is greater than nr and the global bucket level g must rise. The first policer to be updated with a bucket level strictly higher than $G + LT$, will have to wait longer than LT/r time to send its next report. Call this P_1. All other policers have lower b_i value, and with demand over r, all policers will report before P_1 next reports. By induction, each subsequent reporting policer will have to wait an even longer time to report next. This gives a sequence of n unique policers sending reports, with no repeats; we can number these $\{P_1, \ldots, P_n\}$ in order. Now we have the situation above, where as soon as one policer finishes allowing LT traffic, it stops allowing new packets and the next policer starts allowing packets. In other words, P_1 stops once P_2 starts, P_2 stops once P_3 starts and so on, giving us the time-sharing behaviour.

We have complete time-sharing when all $d_i > r$, implicitly requiring $d > nr$. In general, time-staggering will occur between two reporting policers whenever the global bucket goes above $G + LT$, which happens when $d > r$.

3.2.2 Error Bounds on Rate

Any amount of traffic that goes over the desired rate limit is considered the error. Under a centralized leaky bucket algorithm, the error in the amount of traffic permitted (in bytes) is bounded by a constant over time. This bound is called the burst size. Over longer time intervals, the burst becomes insignificant, and the error in the rate itself approaches zero. We can show there is a burst size for the Basic algorithm.

Theorem 4. *The total volume allowed by the Basic algorithm in any given period of time Δt is no more than $r\Delta t + G + nLT$. In other words, the burst size is $G + nLT$.*

Proof. This follows from the maximum global bucket value given in Lemma 2. The burst occurs when we start receiving traffic on all policers when the global bucket is empty, and instantaneously reach the maximum level of $g = nLT + G$. This means that each policer has one stripe (size LT) above the threshold G in the global bucket. The policer with the lowest stripe is at level $b_i = G + LT$. From (3.4) and (3.5), the basic algorithm only permits packets when

$$b_i + LC_i \leq G + LT,$$

so any additional traffic is restricted to rate r and does not add to the burst size. The burst size is $G + nLT$. □

3.2.3 Full Utilization

Like in Padwekar's algorithm[15], the Basic algorithm only restricts traffic based on past overuse, never on future expected use, so the full rate is always used if demand is there. Full utilization is guaranteed, with one requirement of the threshold G, which we prove in the following theorem. Full utilization means that the achieved global rate a must be no less than $\min(d, r)$, where d is the global demand and r the global rate limit.

Lemma 5. *If the total demand is $d \leq r$, the maximum value of $b_i + LC_i$ is nLT for any policer i.*

Proof. Whenever x volume is forwarded at rate d by the entire system, g will have decreased at least x at leak rate r. Any report (or group of reports) that causes the global bucket level to increase, will have been preceded by an even larger decrease, with one exception. The exception is that g can not decrease below 0. The highest level occurs when $g = 0$ initially and all policers update simultaneously, causing $g = nLT$. There is one policer j for which $b_j = g$. Since all policers have updated, $LC_j = 0$. Then this policer has the largest sum of $b_j + LC_j = nLT$. However, since total demand is $d \leq r$, each policer can not possibly increase their local count any faster than their bucket level is decreasing. So nLT is the maximum value of $b_i + LC_i$. □

Theorem 6. *The basic algorithm ensures full utilization iff $G \geq (n-1)LT$.*

Full utilization implies $G \geq (n-1)LT$. Proof. Consider the case where total demand $d = r$. For absolute full utilization, all packets should be allowed. Lemma 5 says the maximum $b_i + LC_i$ can reach is nLT. But if $b_i + LC_i > B$ a packet will be dropped by the algorithm. So for full utilization we require:

$$\begin{aligned} B &\geq nLT \\ G &\geq (n-1)LT \end{aligned} \tag{3.9}$$

$G \geq (n-1)LT$ implies full utilization. Proof. Assume the algorithm does not give full utilization. Then the overall allowed rate a is less than $\min(d, r)$. There are two cases: either $a < d \leq r$ or $a < r < d$.

For the first case, we already know that $b_i + LC_i \leq nLT$ from Lemma 5 and $nLT \leq G + LT = B$ from the premise. Under the policing decision, a packet will never be dropped. This case gives full utilization.

In the other case, on average g must be decreasing, since the leak rate is $r > a$, and this means b_i for all i are also decreasing. Eventually this bucket level will go below $B - LT$, at which point the policing decision will not drop any packets at all. But if no packets are dropped, the total demand $d > r$ would cause g to rise again after several reports. Thus the bucket level can not be decreasing on average, which is a contradiction. □

3.2.4 Unfairness in the Basic Version

In the time-sharing special case from Section 3.2.1, all policers are receiving demand $d_i \geq r$ and all policers get equal r/n rates, which is max-min fair. For general cases, fairness is not always achieved.

A counter-example is when one policer, P_1, receives exactly r/n demand and the rest receive r demand each. The max-min fair allocation is r/n rate each. This can only be achieved if P_1 is permits packets at all times. Instead, when P_1 reports to the master, it is easy to see that it will often receive a high value of b_1. (The high demand policers are receiving traffic fast enough that they can report LT traffic to the master as soon as their previous stripe leaks below threshold. Then the global bucket level will be consistently $G + (n-1)LT$ or higher.) Because the basic algorithm waits until $b_i \leq G + LT$ to start allowing packets, for $n > 1$, P_1 does not allow all packets. This means it allows less than its MMF share of r/n rate, thus giving unfairness. It turns out this waiting is an unnecessary extra restriction.

3.3 Early Start Strategy: Improving Fairness

To address the fairness issues of the Basic algorithm, we propose an "Early Start" strategy. Using this strategy will still comply with the general protocol, *i.e.* a policer can only report LT traffic to the master when the level of its previous report (or stripe) b_i has sunk down below the threshold G. The idea is the policer starts allowing new packets as soon as it reports to the master. If it finishes allowing LT traffic *before* the next allowable report time, then it will both delay reporting to the master and drop all further packets until that time. We can compare four algorithms, two of which use the Early Start strategy:

1. **Basic:** A policer allows LT traffic starting when $b_i \leq G + LT$, at rate r. It reports to the master when $LC_i = LT$, which occurs at earliest when $b_i = G$. This is the basic polling algorithm described in Section 3.2.

2. **Basic, Unrestricted:** A policer allows LT traffic starting when $b_i \leq G + LT$, at unrestricted rate. It reports to the master when $LC_i = LT$, or when $b_i = G$, whichever occurs later. This would allow arbitrarily sharper bursts of traffic, but the time-staggering of the bursts would be the same as the preceding method.

3. **Early Start, Unrestricted:** A policer allows LT traffic starting when the last report is sent, at unrestricted rate. It reports to the master when $LC_i = LT$, or when $b_i = G$, whichever occurs later. This allows traffic through at any arbitrary speed, but stops when LT is reached. Clearly, this maintains the same burstiness and time-staggering as the preceding method, only shifting the occurrence of each burst by an equal amount earlier. Pseudocode for this implementation is given in Algorithm 3.3.

4. **Early Start, Steady:** A policer allows LT traffic starting when the last report is sent, at rate r_i. Choose r_i such that LC_i can accumulate to LT in the same time that it takes b_i to leak to G. We can enforce r_i by using a separate leaky bucket, to give a much steadier output rate. Pseudocode for this implementation is given in Algorithm 3.4.

The following example compares the behaviour of these four algorithms.

Example 1. Let one time unit be the time it takes LT to leak at rate r. Suppose a policer has just reported and received a new bucket level from the master, so that $b_i = G + 2LT$. This specifies that the earliest time for the next report is in 2 time units, *i.e.* the policer can allow up to $r/2$ average rate during the time until the next report.

If this policer is receiving demand at $d_i = 2r$, then it would take $1/2$ a time unit to forward LT traffic if unrestricted. Under the Basic algorithm, it would be allowed to do this in 1 time unit after waiting 1 time unit for $b_i = G + LT$, and under Early Start Steady, 2 time units starting right away. In all variations, the policer does not report until 2 time units have passed. Thus, the average allowed rate achieved is $r/2$ in all cases, and the only difference is in the burstiness. This is depicted graphically in Figure 3.3.

However if instead the policer receives demand at $d_i = r/2$, we no longer get $r/2$ rate in the Basic algorithms. At this demand, it naturally takes 2 time units to allow LT traffic. The basic variations force waiting for 1 time unit, thus finishing in 3 time units total, for an average of $r/3$ rate. With the Early Start strategy, there is no initial waiting, as shown in Figure 3.4. The early start variations allow $r/2$ traffic as desired.

Figure 3.3: Resulting rate achieved by policer receiving $2r$ demand, when the next allowable report time is in 2 time units, on four variations: (a) Basic (b) Basic Unrestricted (c) Early Start Unrestricted (d) Early Start Steady. The area of the shaded area represents volume allowed and equals LT in all cases.

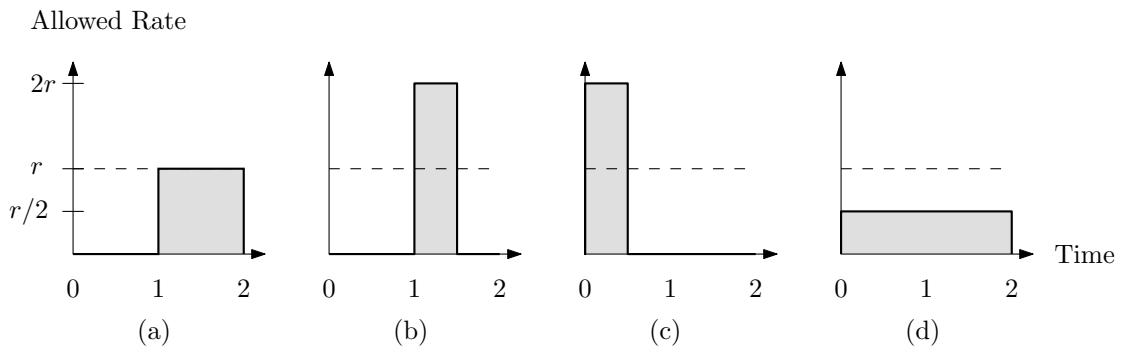
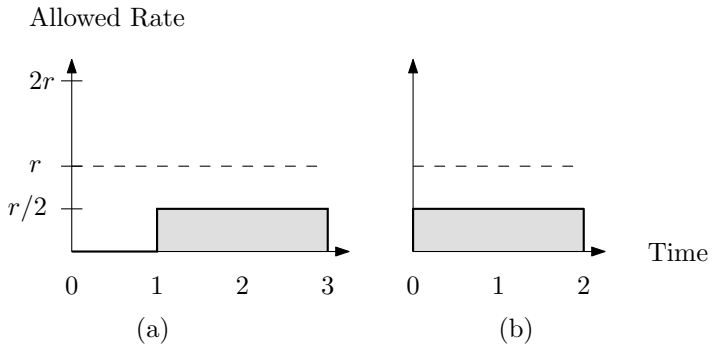


Figure 3.4: Resulting rate achieved by policer receiving $r/2$ demand, when the next allowable report time is in 2 time units, on (a) Basic variations and (b) Early Start variations. The area of the shaded area represents volume allowed and equals LT in all cases.



3.3.1 Implementation

Algorithms 3.3 and 3.4 show how we could implement the Early Start Unrestricted and Early Start Steady algorithms. Only pseudocode for the policers is given; the code for the master remains the same as in Algorithm 3.1.

Like the Basic algorithm, these implementations take the bucket level b_i and continuously decrement it at rate r to mimic the global leaky bucket. However, this variable is no longer used in the policing decision for an incoming regular packet. It only signals when the next report may be sent. Alternatively, we could simply precompute the time at which b_i would leak to G, avoiding unnecessarily changing of the variable. Either way, the policer needs a timing mechanism, so we have presented the first method that leaks b_i.

3.3.2 Error Bounds

As with the Basic algorithm, we can give an error bound on the rate in terms of a burst size. The burst size for both early start variations is $G + 2nLT$, which is slightly larger than $G + nLT$ for the basic algorithm.

Theorem 7. *The total volume allowed by the early start algorithm in any given period of time Δt is no more than $r\Delta t + G + 2nLT$. In other words the burst size is $G + 2nLT$.*

Proof. The burst occurs when we start receiving traffic on all policers when the global bucket is empty, and instantaneously reach the maximum level of $g = G + nLT$ given in Lemma 2. This means that each policer has reported at least once, and each now has one stripe (size LT) above the threshold G in the global bucket. Under the Early Start Unrestricted algorithm, each policer may now allow

Algorithm 3.3 Early Start Unrestricted for policer i

On event: Receiving regular packet

1: **if** $LC_i < LT$ **then**
2: Allow packet
3: $LC_i \leftarrow LC_i +$ packet size
4: **end if**
5: Check condition for reporting

Condition for reporting:

1: **if** $LC_i \geq LT$ **then**
2: **if** $b_i \leq G$ **then**
3: Report(LT) to master
4: $b_i \leftarrow b_i + LT$
5: $LC_i \leftarrow LC_i - LT$
6: **else**
7: (optional) Schedule a report to be sent when $b_i \leq G$
8: **end if**
9: **end if**

another LT bytes in zero time. After this, the rate is restricted to r, because each policer must wait for its stripe of size LT to sink below threshold at rate r. The full burst size is $G + 2nLT$.

□

3.3.3 Full Utilization

The Early Start algorithms presented give full utilization. The proof sketch is that both the Early Start Unrestricted and Early Start Steady variations will allow LT volume in the same amount of time as the Basic algorithm or earlier. Since the basic algorithm gives full utilization (Theorem 6), so do these.

3.3.4 Minimum Rate Guarantee

Using the early start strategy, we can guarantee a minimum rate of r/n for every policer, for every discrete interval between its reports. By guarantee we mean: of the demand received at a policer, we guarantee never to police it below r/n. For the Early Start Unrestricted algorithm, this is always true, no matter how bursty the demand is. Under Early Start Steady, traffic is policed so that the output rate is steady like a leaky bucket, but this steady rate will never be restricted below r/n.

Theorem 8. *The Early Start Unrestricted algorithm imposes no additional restrictions on the rate of a policer, apart from the restriction specified in the general*

Algorithm 3.4 Early Start Steady for Policer i

On event: Receiving update(g) from master

1: $b_i \leftarrow g$
2: **if** $b_i > G$ **then**
3: $r_i \leftarrow r\frac{LT}{b_i - G}$
4: Create new leaky bucket size $E = LT$, with level e and leak rate r_i
5: $e_i \leftarrow LT$
6: **end if**

On event: Receiving regular packet

1: **if** $LC_i < LT$ **and** ($b_i \leq G$ **or** $e_i < E$) **then**
2: Allow packet
3: $e_i \leftarrow e_i +$ packet size
4: $LC_i \leftarrow LC_i +$ packet size
5: **end if**
6: Check condition for reporting

Condition for reporting:

1: **if** $LC_i \geq LT$ **then**
2: **if** $b_i \leq G$ **then**
3: Report(LT) to master
4: $b_i \leftarrow b_i + LT$
5: $LC_i \leftarrow LC_i - LT$
6: **else**
7: (optional) Schedule a report to be sent when $b_i \leq G$
8: **end if**
9: **end if**

protocol in Section 3.1. That is, for any period between reports, with the protocol-specified rate of r_i and demand d_i, the policer i allows packets at a rate of

$$a_i = \min(d_i, r_i) \tag{3.10}$$

when all rates are computed as simple averages (amount divided by time) during the period between reports.

Proof. The general protocol implicitly specifies a rate r_i, during the interval between reports, by restricting the earliest time when the policer may next report. Following the sending of the previous report, the algorithm will allow the first LT traffic that arrives at the policer. Thus, the only way a policer will allow less than r_i is if LT demand does not arrive before the allowable reporting time. Then all packets up to LT will be allowed, and the achieved rate is the same as the demand d_i. The Early Start Unrestricted algorithm never imposes any additional restriction. □

Corollary 9. *The Early Start Unrestricted algorithm never restricts any policer to less than r/n rate over any interval between reports.*

Proof. From Theorem 3, the reporting protocol specifies a rate of r_i between reports that is no less than r/n. Since this algorithm imposes no additional restrictions, any traffic is never policed below r/n. □

Theorem 10. *The Early Start Steady algorithm never restricts any policer to less than r/n leaky bucket rate over any interval between reports.*

Proof. From Theorem 3, the protocol specifies a rate r_i between reports that is no less than r/n. The Early Start Steady algorithm uses a leaky bucket leaking at r_i to allow packets, thus implying the theorem. □

3.3.5 Weighted Rate Guarantees

The analysis of the Early Start strategy allows for a minimum $1/n^{th}$ fraction of the rate to each individual policer. It is possible to give different minimum shares to each policer, if we chose to weight each policer differently.

We next modify the protocol so that each policer still allows a fixed amount of traffic between reports, but the amount is different for each policer, LT_i instead of the same LT for all. The analogy is we still have at most one stripe from each policer in the global bucket above the threshold G, but the size of each stripe is now variable. At capacity, the minimum share of the rate that any policer would get is equal to its fraction of the sum of all LT_i.

Under this technique, the maximum level in the global bucket from Lemma 2 becomes

$$g \leq G + \sum_{j=1}^{n} LT_j \tag{3.11}$$

Then similar to (3.2), the rate specified by the protocol for any policer P_i is

$$\begin{aligned} r_i &= \frac{rLT_i}{b_i - G} \\ &\geq \frac{rLT_i}{G + \sum_j (LT_j) - G} \\ &\geq \frac{rLT_i}{\sum_j (LT_j)} \end{aligned} \tag{3.12}$$

as desired. The two Early Start algorithms we presented would then guarantee not to police below this minimum rate, as described in the previous section.

3.3.6 Almost Max-Min Fair

The rate specified by the general protocol in Equation (3.2), and achieved by the Early Start algorithms, approximates the max-min fair allocation for a policer. It guarantees that any policer will be able to get at least a $1/n^{th}$ share of the rate, which is a requirement for being max-min fair. Intuitively, if any policer uses less than this share, then the global bucket will contain less than the maximum. This would result in lower b_i values upon updating, thus yielding a higher effective rate for the other policers that can use it. The main issue is that the allowed rate depends on the actual bucket level b_i upon updating, and this can vary from one update to the next.

Example 2. Bad Case (Conjectured Worst Case)

We can construct a bad case with policers receiving constant demand. The bad case relies on perfectly coincidental reports by practically all policers, an extreme case to be sure. Let there be $n-2$ policers each receiving low demand, equal to a constant fraction $\alpha \in [0,1]$ of the minimum rate guarantee. So each policer receives:

$$d_{lo} = \frac{\alpha r}{n} < \frac{r}{n} \tag{3.13}$$

Let $\beta \in [0,1]$ be the remaining bandwidth fraction,

$$\begin{aligned} \beta &= 1 - \frac{(n-2)\alpha}{n} \\ &= \frac{n - n\alpha - 2\alpha}{n} \end{aligned} \tag{3.14}$$

Of the other two policers, one (P_{mid}) has demand equaling half the remaining bandwidth, which exceeds the minimum rate guarantee:

$$d_{mid} = \frac{\beta r}{2} > \frac{r}{n}$$

The last (P_{hi}) is receiving demand

$$d_{hi} \gg r$$

The max-min fair allocation is to give the low and mid demand policers all their demand, with the high demand policer getting the same as the mid.

Since each low policer's demand is less than r/n, the algorithm is obligated to allow all their packets. By setting them all to the same constant rate, all $n-2$ local counts will fill and send simultaneous updates. Meanwhile, the high demand policer will quickly cause the bucket level to rise above G (this is a required property to achieve full utilization). Afterwards, the high policer will always have one stripe in the global bucket. It is clear that upon the arrival of $n-2$ simultaneous reports, the bucket level will be over threshold by a $n-1$ stripes. It can be shown that P_{mid} will have to report at some point while the bucket level is still high, and that it will finish allowing LT traffic before its stripe drains. Then it must drop packets, failing to achieve the max-min fair allocation.

Intuitively, the unfairness is caused by the variation between high bucket level (when low policers all update) and low bucket level (after bucket leaks down, but before low policers have enough to send an update). At high level, P_{mid} and P_{hi} are restricted to $\sim r/n$. At low level, they are only restricted to $\sim r/2$, but P_{mid} is unable to use all this rate. P_{hi} uses up what P_{mid} can not use (due to efficiency).

3.3.7 Bad Case Analysis

Let us analyze the bad case above for P_{mid}. (We conjecture this is the worst case when all demands are steady.) The demands are set up so that the $n-2$ policers always report at the same time. The maximum global bucket level of $G+nLT$ from (3.1) is reached when all n policers' reports coincide. Assume this happens and the mid policer's report is the last to arrive at the master, so that P_{mid} receives the maximum level of b_{mid}. The time this takes to leak down to G is nLT/r. The next group of P_{lo} reports occurs in $nLT/r\alpha$ time. Also assume that at each such group of reports, P_{mid} arrives at the same time, always as the last one.[2] Therefore for α of the time, P_{mid} is restricted to r/n rate. In this case, the achieved rate fraction for P_{mid} is:

$$\begin{aligned}\frac{a_{mid}}{r} &= \frac{\alpha}{n} + \frac{(1-\alpha)\beta}{2} \\ &= \frac{(n-2)\alpha^2}{2n} - \frac{(n-2)\alpha}{n} + \frac{1}{2}\end{aligned} \tag{3.15}$$

We can verify this equation. If $n=2$ or $\alpha=0$, Equation (3.15) says $1/2$ the rate limit is given to P_{mid}. This makes sense because if $n=2$, then there is only one mid and one high demand policer, and the P_{mid} asks for, and is guaranteed, exactly $1/2$ the rate. If $\alpha=0$, the low demand policers ask for no demand and have no effect on the system. Again, since only 2 policers ever report to the master, the global bucket is at most $G+2LT$ and $r/2$ rate is guaranteed.

[2]This is not always the case, but such cases do exist and are simple to construct. Such cases are presented and simulated in Chapter 4.

On the other hand, if $\alpha = 1$, Equation (3.15) gives $1/n$ rate fraction. Note $\alpha = 1$ implies the low demand policers have demand of r/n which is no longer "low demand" by our definition. This means P_{mid} has demand $d_{mid} = r/n$ as well, which is guaranteed. Again, this matches the equation.

To be max-min fair, P_{mid} should be allowing all of its traffic demand. To judge how far from fair the algorithm is, we can look at either the absolute or relative difference from actual to fair allocation.

Fairness Ratio

Let us look at the ratio of what P_{mid} gets to the fair share:

$$\begin{aligned} \frac{a_{mid}}{d_{mid}} &= \frac{2\alpha}{n\beta} + (1-\alpha) \\ &= \frac{2\alpha}{n - n\alpha + 2\alpha} + (1-\alpha) \end{aligned} \tag{3.16}$$

We can quickly verify this equation as well. If $\alpha = 1$ or $\alpha = 0$ or $n = 2$ the ratio is 1, which matches the cases described above because P_{mid} is getting the fair share in each case.

Try to find the worst fairness ratio by finding the value of α that minimizes Equation (3.16). Take the derivative with respect to α:

$$\frac{2(n - n\alpha + 2\alpha) - 2(-n+2)\alpha}{(n - n\alpha + 2\alpha)^2} - 1 \tag{3.17}$$

Set this to zero.

$$\begin{aligned} 2(n - n\alpha + 2\alpha) + 2(n-2)\alpha &= (n - n\alpha + 2\alpha)^2 \\ 2n &= n^2 - 2n^2\alpha + 4n\alpha - 4n\alpha^2 + n^2\alpha^2 + 4\alpha^2 \\ 0 &= (-4n + n^2 + 4)\alpha^2 + (-2n^2 + 4n)\alpha + (n^2 - 2n) \\ 0 &= (n-2)^2\alpha^2 + 2n(2-n)\alpha + n(n-2) \\ 0 &= (n-2)([n-2]\alpha^2 - 2n\alpha + n) \end{aligned} \tag{3.18}$$

Note that if $n = 2$ then the derivative is 0 at all values of α. This is because the ratio equals 1 when $n = 2$. For $n > 2$, the critical values of α are the solutions to:

$$0 = [n-2]\alpha^2 - 2n\alpha + n \tag{3.19}$$

Using the quadratic formula:

$$\begin{aligned}\alpha &= \frac{2n \pm \sqrt{4n^2 - 4(n-2)n}}{2(n-2)} \\ &= \frac{2n \pm \sqrt{4n^2 - 4n^2 + 8n}}{2n-4} \\ &= \frac{2n \pm \sqrt{8n}}{2(n-2)} \\ &= \frac{n \pm \sqrt{2n}}{n-2} \end{aligned} \tag{3.20}$$

Given that $n > 2$ and $\alpha \in [0, 1]$, then $\frac{n+\sqrt{2n}}{n-2} > 1$ is an extraneous solution. The only appropriate value of α is $\frac{n-\sqrt{2n}}{n-2}$ because this value is $\in (0, 1)$.

Proof.

$$\begin{aligned} n &> 2 \\ 2n &> 4 \\ \sqrt{2n} &> 2 \\ n - \sqrt{2n} &< n - 2 \\ \frac{n - \sqrt{2n}}{n-2} &< 1 \end{aligned} \tag{3.21}$$

Also,

$$\begin{aligned} n &> 2 \\ n^2 &> 2n \\ \sqrt{n^2} &> \sqrt{2n} \\ n &> \sqrt{2n} \\ n - \sqrt{2n} &> 0 \end{aligned} \tag{3.22}$$

$$\frac{n - \sqrt{2n}}{n-2} > 0 \tag{3.23}$$

□

So the value of α minimizing Equation (3.16) is

$$\alpha_{\min} = \frac{n - \sqrt{2n}}{n-2} \tag{3.24}$$

The worst case ratio is

$$\left(\frac{a_{mid}}{d_{mid}}\right)_{\min} = \frac{2\alpha_{\min}}{n - n\alpha_{\min} + 2\alpha_{\min}} + (1 - \alpha_{\min})$$

which simplifies to:

$$= \frac{2\sqrt{2}}{\sqrt{n} + \sqrt{2}} \tag{3.25}$$

which can get arbitrarily small as n grows large. There is no value of n that minimizes the ratio. Although this can make it very unfair for P_{mid}, we note that increasing n also increases the number of policers getting exactly the fair share, $n-2$.

Absolute Fairness Difference

We can also look at the absolute difference between the achieved rate and the fair rate. In this case we find a $r/8$ difference at worst in the bad case above.

Since d_{mid} is the fair share for the mid policer, the difference between the fair and achieved shares is:

$$\begin{aligned}\frac{d_{mid}}{r} - \frac{a_{mid}}{r} &= \frac{\beta}{2} - \left(\frac{\alpha}{n} + \frac{(1-\alpha)\beta}{2}\right) \\ &= \frac{n\alpha - n\alpha^2 + 2\alpha^2 - 2\alpha}{2n} \\ &= \frac{(n-2)\alpha - (n-2)\alpha^2}{2n}\end{aligned} \tag{3.26}$$

To verify, note that if $\alpha = 0$ or $\alpha = 1$ or $n = 2$, the difference is 0, *i.e.* it is max-min fair, which agrees with our previous analysis.

Take the derivative with respect to α and set it equal to zero:

$$\begin{aligned}\frac{n-2}{2n} - \frac{2(n-2)}{2n}\alpha &= 0 \\ \frac{n-2}{2n} &= \frac{2(n-2)}{2n}\alpha \\ \alpha &= 1/2\end{aligned} \tag{3.27}$$

Thus $\alpha = 1/2$ maximizes the fairness difference. The largest fairness difference is then:

$$\begin{aligned}\frac{(n-2)(1/2)(1/2)}{2n} &= \frac{n-2}{8n} \\ &\leq \frac{1}{8}\end{aligned} \tag{3.28}$$

The bad case fairness difference value is no worse than $1/8$ the rate as n approaches infinity. In practical terms, this means as n grows large, the worst fairness difference occurs when the low demand policers together take up $1/2$ the bandwidth, leaving $\beta = 1/2$. The max-min fair allocation is for the mid and high policers to each receive $1/4$ the bandwidth, but instead the mid gets $1/8$ and the high gets $3/8$ under the Early Start algorithms.

3.3.8 A Partial Solution: Seeding

One way to alleviate the bad case affecting the Early Start algorithms is to seed the initial local counts. The policers' LC_i values are initialized with random or uniformly spaced values between 0 and LT (or $-LT$ and $+LT$) instead of 0. This avoids the coincidental occurrence of reports by policers that receive demand at the exact same rate, because each policer now fills to $LC_i = LT$ at a different time. Note that we retain the same theoretical guarantees on minimum rates, error bounds, and utilization; only the initial reporting time will change. In doing so, it greatly reduces the chance that many policers report at once, even in cases where demands are equal or have very small lowest common multiples.

The seeding strategy does not give us any better guarantees. It is possible that the policers start receiving demand at different times, and if the starting times were chosen to exactly nullify the seeded local counts, we would have reconstructed the bad case. This is highly unlikely even if an adversary picks the start times, as long as the adversary does not know the exact initial values used. We stress that because the LC_i values are not leaked, there is no chance of "idling" until they reach zero; the start times must be picked exactly.

We have simulated the seeding strategy on the bad case, when all demands are steady and start at the same time. In the experiments, seeding brings the Early Start algorithms much closer to max-min fairness. We present these results in Chapter 4.

3.4 Low Usage Credit

The Low Usage Credit modification is both an algorithm for the policer and a modification to the general reporting protocol. Since the Early Start Unrestricted algorithm is already the most flexible way to comply with the protocol, this modification deviates from the protocol's requirements in an attempt to achieve better fairness. However by doing so, we lose several nice theoretical properties. As such, this modification is considered experimental, but it does perform better in the bad cases affecting the Early Start and Basic algorithms.

As with the general protocol, the b_i value received from the master specifies the next allowable report time. However *a policer is allowed to violate this and report earlier, if in a previous interval it had reported later than the allowable time.* The idea is to average out the fluctuations of high and low values received, which was the main problem causing the bad case for Early Start. The master is not aware of the violation and operates the same as the general algorithm. Only the policer's behaviour is changed.

When sending a report, each policer checks to see if it is doing so later than the allowable time. This can be measured by the policer's b_i value compared to G. If the report occurs later than necessary, b_i will have leaked below G, and this

difference is saved in a credit counter, CR_i, per policer. This credit is cumulative. The credit allows a policer to report earlier than allowed as follows: The protocol originally specifies a report can be sent when $b_i \leq G$. With the credit counter, the time a policer may report is changed to when:

$$b_i \leq G + CR_i$$

When positive credit exists, then it is possible for a policer to report when $b_i > G$. The difference in values can be subtracted from CR_i. Thus, credit is awarded when a policer's usage is lower than its given share, and it can be redeemed when a policer needs higher usage than its share.

3.4.1 Implementation

The Low Usage Credit modification can be used in combination with the Early Start modification, but it is not strictly necessary. It can be applied to the Basic algorithm, and we observe that this is enough to give fairness improvements in cases that were problematic for the Basic algorithm alone. The pseudocode is presented in Algorithm 3.5. Like the Basic algorithm, we set a leaky bucket threshold $B = G + LT$.

In Algorithm 3.6, the pseudocode for modifying Early Start Unrestricted with the Low Usage Credit is shown.

Algorithm 3.5 Basic + Low Usage Credit for policer i

On event: Receiving update(g) from master

1: $b_i \leftarrow g$

On event: Receiving regular packet

1: **if** $b_i + LC_i < B + CR_i$ **then**
2: Allow packet
3: $LC_i \leftarrow LC_i +$ packet size
4: Check condition for reporting
5: **end if**

Condition for reporting:

1: **if** $LC_i \geq LT$ **then**
2: Report(LT) to master
3: $CR_i \leftarrow CR_i + G - b_i$
4: $b_i \leftarrow b_i + LT$
5: $LC_i \leftarrow LC_i - LT$
6: **end if**

Algorithm 3.6 Early Start Unrestricted + Low Usage Credit for policer i

On event: Receiving update(g) from master

1: $b_i \leftarrow g$

On event: Receiving regular packet

1: **if** $LC_i < LT$ **then**
2: Allow packet
3: $LC_i \leftarrow LC_i +$ packet size
4: **end if**
5: Check condition for reporting

Condition for reporting:

1: **if** $LC_i \geq LT$ **then**
2: **if** $b_i \leq G + CR_i$ **then**
3: Report(LT) to master
4: $CR_i \leftarrow CR_i + G - b_i$
5: $b_i \leftarrow b_i + LT$
6: $LC_i \leftarrow LC_i - LT$
7: **else**
8: (optional) Schedule a report to be sent when $b_i \leq G + CR_i$
9: **end if**
10: **end if**

3.4.2 Wait Time is Reduced

The Low Usage Credit modification does not require the use of the Early Start modification to be effective. A policer that has credit is allowed to report earlier, and this implicitly means the policer may start allowing packets earlier under the Basic + Low Usage Credit algorithm (Algorithm 3.5). Under steady rate demands, we find that this algorithm allows a policer to start allowing packets just early "enough."

Let a policer be receiving steady demand at rate d_i. Let $\hat{d}_i = \min(d_i, r)$ be its demand capped at r, so that $0 < \hat{d}_i \leq r$. The amount of time it takes this policer to permit LT traffic is always:

$$\Delta t_{\text{allow}} = \frac{LT}{\hat{d}_i} \tag{3.29}$$

Assume the credit is 0 and the policer receives a value of $b_i > G + LT$ from the master. Under the Basic + Low Usage Credit algorithm, it only starts permitting traffic when $b_i \leq G + LT$, so the policer's bucket level just before reporting is:

$$\begin{aligned} b_i &= G + LT - r\Delta t_{\text{allow}} \\ &= G + LT - \frac{rLT}{\hat{d}_i} \end{aligned} \tag{3.30}$$

From line 3 of the condition to send an update (Algorithm 3.5), the credit becomes

$$\begin{aligned} CR_i &\leftarrow CR_i + G - b_i \\ &= 0 + G - (G + LT - \frac{rLT}{\hat{d}_i}) \\ &= \frac{rLT}{\hat{d}_i} - LT \end{aligned} \tag{3.31}$$

The policer then receives the updated b_i from the master, which determines how long the policer needs to wait before allowing the next packet. Assuming $b_i \geq B + CR_i$, it needs to wait until $b_i = B + CR_i$. If any credit was collected from the last report, this wait time is reduced. The time it spends dropping packets (*i.e.* the wait time) is:

$$\begin{aligned} \Delta t_{\text{drop}} &= \frac{b_i - (B + CR_i)}{r} \\ &= \frac{b_i - G - LT - \frac{rLT}{\hat{d}_i} + LT}{r} \\ &= \frac{b_i - G}{r} - \frac{LT}{\hat{d}_i} \end{aligned} \tag{3.32}$$

Now if we look at the total time taken between reports, which is the sum of time to allow plus time to wait, we find that the total time does not depend on the policer's demand $\hat{d}_i$.

$$\begin{aligned}\Delta t_{\text{allow}} + \Delta t_{\text{drop}} &= \frac{LT}{\hat{d}_i} + \frac{b_i - G}{r} - \frac{LT}{\hat{d}_i} \\ &= \frac{b_i - G}{r}\end{aligned} \tag{3.33}$$

Compare with the Basic algorithm where a smaller $\hat{d}_i$ means longer time overall:

$$\Delta t_{\text{allow}} + \Delta t_{\text{drop}} = \frac{LT}{\hat{d}_i} + \frac{b_i - (G + LT)}{r} \tag{3.34}$$

Additionally, the time given in equation 3.33 is exactly the amount of time in which the bucket level b_i leaks to G, hence the credit counter CR_i is unchanged. Thus we have arrived at a stable equilibrium.

This analysis shows that the Low Usage Credit reduces the packet drop time in response to low demand, so that the policer can start allowing packets just early enough to report in time. Thus, the Early Start strategy is not necessary when using Low Usage Credit. Assuming all policers receive constant and unchanging demand, the time between reports is purely determined by the updated bucket level b_i and not the policer's own demand, thus eliminating the bias towards high demand policers exhibited by the Basic algorithm.

However, our assumption that the updated b_i value is larger than $B + CR_i$ is not always true. If not, the algorithm allows packets right away without dropping any packets, and the policer will report later than necessary, due to low demand. This is the same behaviour as the Early Start algorithms, except the credit counter will remember this under-usage indefinitely, while Early Start forgets it at each new reporting interval.

3.4.3 Issues and Error Bounds

Since the Low Usage Credit modification allows violation of our general reporting protocol, it loses many of the theoretical guarantees provided by that protocol. Most notably, Lemma 2 relating to the maximum level of the global bucket and Theorem 3, which leads to the r/n minimum rate guarantee, both do not hold. However, since it is still a reporting system, the no-starvation guarantee does hold, and we conjecture full utilization holds as well. We are also able to give an error bound on the rate, which is quite different from before. In general, the carry-over of credit makes analysis difficult relative to the Basic and Early Start algorithms. It may be possible to refine the analysis, but we treat this as more of an experimental algorithm.

A policer that is receiving traffic demand at low rate can accumulate unbounded credit under the Low Usage Credit modification. Under our simple case studies and simulations where demands are constant, this has not been a problem. However, if demands skyrocket, the policer with large credit will allow a large burst of traffic. We should bound the maximum credit any policer can accumulate to some value $CR_{\max}$. Interestingly, the credit is consumed quadratically.

Claim 11. *The maximum burst size for the Basic + Low Usage Credit algorithm is*

$$\frac{(-1+\sqrt{1+8nCR_{\max}})LT}{2} + G + nLT \tag{3.35}$$

Proof. The largest burst size occurs when the global bucket is empty ($g = 0$), all n policers have maximum credit, and all policers receive infinite demand. The first G/LT reports do not consume nor increase the credit (because credit is already maximized), but push the bucket level to G. Each of the n policers still has a b_i value less than or equal to G, and can each report once more without consuming credit. After this, each subsequent report sent by a policer will decrease that policer's credit by $b_i - G$. Since the global bucket increases by one LT at each report, the first report consumes LT, the second consumes $2LT$, etc. until $nCR_{\max}$ credit is expended. There are at most k reports that consume credit, where

$$\begin{aligned} 1 + 2 + 3 + 4 + \cdots + k &= nCR_{\max} \\ \frac{k(k+1)}{2} &= nCR_{\max} \end{aligned} \tag{3.36}$$

The solution is $k = \frac{(-1+\sqrt{1+8nCR_{\max}})}{2}$ and the maximum burst size is approximately

$$\frac{(-1+\sqrt{1+8nCR_{\max}})LT}{2} + G + nLT$$

□

3.5 Other Considerations

Communication Loss and Latency. If an update is lost from master to policer, the policer may report traffic earlier than allowed. Lemma 2 would temporarily not hold and the error could exceed the bound. However, the policer will recover the next time it reports to the master, restoring the correct burst size. If a report is lost from policer to master, then the overall traffic allowed can be permanently LT above the upper bound. However, it has been suggested that correcting for lost communication messages could be implemented at the master if each policer reports the cumulative amount of traffic allowed, rather than only the amount allowed in each interval. Alternatively, each outgoing report from a policer could be numbered in sequence. In both cases, the master would keep the last received value from each policer, so that it can detect and account for missing reports.

In case of such communication packet loss, or even high latency, we do not want to block a policer from allowing packets while it waits for a message from the master. At the same time, we do not want to let it permit traffic without bound. A policer that is sending a report of size LT knows that the master bucket level must increase by *at least* LT. Then it can increment b_i by LT before reporting. Once the master's update is received, it can properly synchronize the bucket level. In case of a series of communication packet losses from master to policer, this prevents the individual policer from going over r rate limit by itself. All of the pseudocode presented for our algorithms use this method.

Uneven Packet Sizes. In all of the analyses, we have assumed that the local count would accumulate packets up to exactly LT bytes and then report. In actuality, it may be that $LC_i > LT$ upon triggering a report. If the policer reports the LC_i value, this would give slight bias towards policers that reach higher LC_i. We eliminate this bias by specifying that all reports increase the global bucket level by LT, and the policer's LC_i is decremented by LT instead of zeroed. All of the pseudocode presented for our algorithms use this method.

Fairness Per Flow. Our method could be extended to give rate guarantees between flows, simply by creating a policer for each flow. This would require space linear in the number of flows as well as knowing the maximum number of flows beforehand to properly set G. This space requirement is no worse than Stanojević and Shorten's virtual queues per flow, used in the D2R2 algorithm [19].

Chapter 4

Simulation Results

4.1 Methods

To verify our algorithms, we used a simulator developed while working with Cisco Systems. It is a simple event-based simulator of packets arriving at a number of policers in a distributed policing system. Other details, such as routing or where the packet goes after leaving the policer, are abstracted out. We have only simulated steady traffic patterns, *i.e.* evenly-spaced packet arrival times. The tests here are by no means exhaustive; however they should support our hypotheses of worst case fairness scenarios.

4.2 Typical Cases

The majority of experiments that we ran showed the Early Start and Low Usage Credit algorithms achieving very close to max-min fair rates between policers, while the Basic algorithm was insufficient. Here we show the results from two such experiments.

The simulations were run with the following parameters. Note that the packet size is constant; traffic demand rates were varied by changing the time interval between packet arrivals.

- Policers: $n = 4$
- Packet size: 10 bytes
- Rate Limit: $r = 10000$ packets/sec $= 100000$ Bps
- Local threshold: $LT = 100$ bytes
- Global Bucket threshold: $G = (n-1)LT = 300$ bytes

Table 4.1: Results for Basic, Early Start Unrestricted, Low Usage Credit algorithms, $n = 4$ policers, typical experiment 1. The ideal shares are max-min fair (MMF).

Policer	Demand (% of r)	MMF (%)	Achieved rate (% of r) Basic	ESU	LUC
1	50	26 2/3	29.42	26.67	26.67
2	40	26 2/3	29.41	26.67	26.67
3	30	26 2/3	23.53	26.67	26.67
4	20	20	17.65	20.00	20.00
Total	140	100	100.01	100.01	100.01

Table 4.2: Results for Basic, Early Start Unrestricted, Low Usage Credit algorithms, $n = 4$ policers, typical experiment 2. The ideal shares are max-min fair (MMF).

Policer	Demand (% of r)	MMF (%)	Achieved rate (% of r) Basic	ESU	LUC	ESU+LUC
1	100	32.5	40.01	32.51	30.02	32.51
2	35	32.5	30.00	32.50	35.00	32.50
3	25	25	20.00	25.00	25.00	25.00
4	10	10	10.00	10.00	10.00	10.00
Total	170	100	100.01	100.01	100.01	100.01

- Simulation time: 60 seconds

The first experiment was performed with the demands at each policer listed in Table 4.1. The results show that the Basic algorithm fails to achieve the ideal (max-min fair) allocation, as policers 3 and 4 do not get the rates they are entitled to. Both the Early Start Unrestricted (ESU) and Basic with Low Usage Credit (LUC) algorithms achieve the ideal results, with less than 0.01% error.

The second experiment was performed with the demands at each policer listed in Table 4.2. Similarily, the Early Start Unrestricted (ESU) algorithm gives max-min fairness, and the Basic does not. This was one of the few examples where the Basic with Low Usage Credit (LUC) algorithm performed worse than ESU. Policer 2 with lower demand actually receives a larger share than policer 1 with a higher demand. We noticed that in this scenario, policer 2 collected much credit, while policer 1 collected none. Under the same setup, but changing policer 1's demand rate to above 110% or below 90%, the Basic with Low Usage Credit algorithm was able to reach MMF. Otherwise, combining the Early Start Unrestricted with Low Usage Credit (ESU+LUC) algorithm also gave good results.

4.3 Unfair Cases for Early Start Algorithm

In this section, we show empirical results from simulations which motivated our bad case analysis in Section 3.3.7. Since the unfair cases rely on a large number of

Table 4.3: Unfair example under Early Start Unrestricted algorithm, $n = 10$ policers, $\alpha = 0.5$.

Policer		Demand		Actual
Type	Count	(% of r)	MMF (%)	(% of r)
lo	8	5	5	5
mid	1	30	30	20
hi	1	100	30	40
total	10	170	100	100

policers sending reports at the same time, we use a priority system at the master that explicitly accepts certain policers' reports before others that arrive at the same time. To find the worst results, we gave the "mid" policer the lowest priority. The following results for the Early Start Unrestricted algorithm were simulated with the following parameters:

- Policers: $n = 10$
- Packet size: 40 bytes
- Rate Limit: $r = 2500$ packets/sec $= 100000$ Bps
- Local threshold: $LT = 400$ bytes
- Global Bucket threshold: $G = (n-1)LT = 3600$ bytes
- Simulation time: 60 seconds

We found a particularly unfair case with the following demands in Table 4.3, which gives a significant discrepancy between the MMF share and actual share for one single policer (labeled "mid"). To construct this bad case, there are 8 low demand policers each receiving demand at $\alpha = 0.5$ times the minimum rate guarantee, *i.e.* $0.5r/n = r/20$ or 5% of the rate. To be max-min fair, all these policers should be allowed their full demand. Together they take up 40% of the bandwidth, leaving 60% available. A high demand policer is receiving demand at r or 100% of the rate and a mid demand policer receives demand at 30% the rate. To be max-min fair, each of these two should equally be allowed 30%, but they are not. Instead the mid demand policer, which is "playing nice" by only asking what it deserves, gets only 20% of the rate. (This is exactly what is predicted by Equation (3.15) in Section 3.3.7.) The mid policer only achieves 2/3 the max-min fair share, and 1/2 of what the high demand policer gets.

We also tried adjusting the mid policer demand, but keeping the low demand and high demand constant. The results are shown in Table 4.4. We found the worst behaviour when the mid policer is asking exactly what it deserves (30%). At higher demands, the MMF share is still 30% but there is more traffic to compete with P_{hi}'s demand. At 50% demand and higher, the mid policer is fully able to

Table 4.4: Resulting rates when varying mid policer demand under Early Start Unrestricted algorithm, $n = 10$ policers, $\alpha = 0.5$.

d_{mid} (% of r)	MMF$_{mid}$ (% of r)	a_{mid} (% of r)
10	10	10.00
15	15	14.99
20	20	15.01
25	25	20.00
30	30	20.27
35	30	24.64
40	30	25.02
45	30	28.51
50	30	30.02

compete with the high demand policer (100%) and gets its fair share. At lower demands, the MMF share is still equal to what it asks for, but we get closer to the minimum guaranteed rate (10%). Oddly, the achieved rates do not scale linearly between these points.

4.3.1 Varying Low Demand α

For the next series of tests, we increased the number of policers to 100 ($n - 2 = 98$ low demand, 1 mid demand, 1 high demand). This setup magnifies the fairness discrepancy.

- Policers: $n = 100$
- Packet size: 10 bytes
- Rate Limit: $r = 10000$ packets/sec $= 100000$ Bps
- Local threshold: $LT = 40$ bytes
- Global Bucket threshold: $G = (n - 1)LT = 3600$ bytes
- Simulation time: 60 seconds

Running an analogous experiment to that of Table 4.3, we have 98 low demand policers receiving demand at $\alpha = 0.5$ times the minimum rate guarantee, *i.e.* $0.5r/n = r/200$ or 0.5% of the rate. Together they take up 49% of the bandwidth, leaving 51% available. A high demand policer is receiving demand at r or 100% of the rate and a mid demand policer receives demand at 25.5% the rate (half the available bandwidth). The results are shown in Table 4.5. Now, the mid policer only achieves slightly over half the max-min fair share, and roughly 1/3 of what the high demand policer gets.

Table 4.5: Unfair example under Early Start Unrestricted algorithm, $n = 100$ policers, $\alpha = 0.5$.

Policer Type	Policer Count	Demand (% of r)	MMF (%)	Actual (% of r)
lo	98	0.5	0.5	0.5
mid	1	25.5	25.5	13.3
hi	1	100.0	25.5	37.8
total	100	174.5	100.0	100.1

We now look at the effects of varying the demand d_{lo} for each low demand policer. The low policer demand is set to $\alpha r/n$, varying α between 0.01 and 0.99. The mid policer's demand d_{mid} is set to half the remaining bandwidth each time, so that its max-min fair share MMF_{mid} is the same as its demand. The results are shown in Table 4.6. The value $\alpha = 0.5$ gives the greatest difference between MMF_{mid} and a_{mid}, but somewhere between $\alpha = 0.75$ and $\alpha = 0.90$ is the worst fairness ratio; the allotted rate is less than 1/3 the MMF rate. Equations (3.24)-(3.25) predict the worst case ratio occurs at $\alpha = 0.876$ and gives 0.2478 ratio, or roughly 1/4. Experimental results are slightly better than the equation predicts though.

Table 4.6: Resulting rates when varying low policer demand (α) under Early Start Unrestricted algorithm, $n = 10$ policers. All demands are in % of r.

d_{lo} (each)	Predicted worst	a_{mid}	d_{mid} = MMF$_{mid}$	a_{hi}
0.01	49.0	48.9	49.5	50.2
0.10	40.7	40.7	45.0	49.6
0.25	28.6	28.6	37.5	47.0
0.33	23.0	23.1	33.9	44.7
0.50	13.3	13.3	25.5	37.8
0.66	6.7	6.8	17.7	28.7
0.75	4.1	4.4	13.2	22.1
0.90	1.5	1.8	5.9	10.1
0.99	1.0	1.0	1.5	2.1

4.3.2 Improvements from Seeding

In Section 3.3.8, we noted that seeding can be used to combat the specific unfair case affecting the Early Start algorithms. Our experimental results show that it can indeed improve the fairness, but does not completely reach max-min fairness. We start with the same setup as the previous section. We noticed that the size of the local threshold (reporting traffic size) affected the effectiveness of the seeding, so we tried a few different setups. These settings were in common across the tests:

Table 4.7: Comparing seeding on Early Start Unrestricted algorithm, $\alpha = 0.5$. Rates shown for P_{mid} and are in % of r.

n	LT	MMF	Predicted worst	Type of Seeding None	Uniform	Random
10	40	30	20	20.00	25.00	23.33 - 26.67
10	400	30	20	20.03	27.50	25.00 - 30.00
10	4000	30	20	20.25	27.54	25.09 - 29.99
100	40	25.5	13.25	13.28	14.01	14.01 - 14.51
100	400	25.5	13.25	13.58	25.00	21.54 - 22.54

- Packet size: 10 bytes
- Rate Limit: $r = 10000$ packets/sec = 100000 Bps
- Simulation time: 60 seconds

These settings were varied:

- Policers: $n = 10$ or 100
- Local threshold: $LT = 40$ or 400 or 4000 bytes
- Global Bucket threshold: $G = (n-1)LT$

We seeded the local counts LC_i of each policer using two methods: 1) uniformly spaced values between 0 and LT, and 2) random values between 0 and LT. The resulting rates for the mid policer are shown in Table 4.7. For the random method, we ran the test 10 times and presented the range. Without seeding, the Early Start algorithms perform very close to the predicted worst case allocation given in Equation (3.15). Both seeding methods improved the share, with the random seeding sometimes worse and sometimes better than the uniform seeding. There was one exception where all random trials gave consistently less fair shares than the uniform seeding. We are unable to explain this behaviour. Increasing the size of the local count threshold LT improved the benefit from seeding. This effect was especially significant in the 100 policers case. We speculate there is greater variation in policer reporting times when LT is higher. In the case where $LT = 40$, only 4 packets of size 10 are allowed before a policer reports, and so any seeded LC_i value will pigeonhole a policer into one of 4 possible reporting slots.

4.3.3 Improvements from Low Usage Credit

Finally we ran the same experiments as the previous section on our other experimental modification, Low Usage Credit. We tried both the Basic + Low Usage

Table 4.8: Comparing different versions of Low Usage Credit algorithm and seeding, $\alpha = 0.5$. Rates shown for P_{mid} and are in % of r.

n	LT	MMF	Predicted worst	LUC	LUC Seeding	ESU+LUC	ESU+LUC Seeding
10	40	30	20	25.00	28.00	23.34	29.99
10	400	30	20	24.99	29.98	24.99	29.98
10	4000	30	20	24.89	29.79	24.89	29.79
100	40	25.5	13.25	13.52	15.70	13.52	16.00
100	400	25.5	13.25	13.89	25.18	13.65	25.18

Credit (LUC) and Early Start Unrestricted + Low Usage Credit (ESU+LUC) algorithms with and without uniform seeding. Random seeding was not attempted. The resulting rates for P_{mid} are shown in Table 4.8. Notably the LUC algorithm gave better fairness in the 10-policer tests, but not the 100-policer test. In all cases, seeding generally improved the result, whereas the combination of ESU+LUC was rarely better than LUC alone.

Note that the implementations tested did not limit policers to a maximum credit, as suggested in Section 3.4.3. In almost all cases, this resulted in accumulation of very large amounts of credit for all the low and mid demand policers. This was not a problem in the tests because all demands were constant.

Chapter 5

Conclusion

5.1 Summary

The distributed policing problem has practical applications in multiprocessor networking devices such as routers and switches, as well as servers in a cloud-computing service. In all of these applications, a customer pays for, and is entitled to, a specified rate. We have looked at this problem prioritizing the full utilization aspect for the customer.

Most prior solutions to the distributed problem are predictive algorithms that adjust rates using either feedback mechanisms or demand estimation, and their outputs only asymptotically converge to the desired rate. These solutions do not guarantee full utilization or any error bounds during the adjustment period because each policer has incomplete information about the system. Padwekar's solution is a non-predictive synchronization algorithm that tracks exact usage, giving error bounds and full utilization, but has several shortcomings, including possibility of starvation, burstiness, and high overhead.

We have presented a general protocol where policers report to a master (or global bucket) such that no information is lost. The solution is similar to that of a FIFO ordering problem, and can be described as a polling system. The protocol has several nice properties and allows for easy derivation of error bounds. However, there are many ways that a policer algorithm could handle packets and still follow the protocol; we have suggested several.

The Basic algorithm we presented is very similar to Padwekar's algorithm, but with a different update system that requires strictly less overhead (lowered from $n-1$ updates per LT (or fewer) bytes usage, to 2 updates per LT bytes usage exactly). It also proves starvation is impossible and full utilization is guaranteed. However, we find the Basic algorithm to be unnecessarily restrictive, and is unable to achieve max-min fairness in even trivial cases.

The Early Start Unrestricted and Early Start Steady algorithms are the least restrictive and give the best guarantees. Each individual policer is essentially guar-

anteed a $1/n^{th}$ share of the rate, with higher shares occurring when global usage is low. Every discrete chunk of LT traffic that a policer P_i allows is spread out over a time period, such that the allowed rate averages $\min(r_i, d_i)$ where $r_i \geq r/n$ is given by the protocol. This heuristically approaches max-min fairness. In analysis and simulations, these variations do reach the ideal max-min fair allocations on many test cases, but fall short on several constructed cases.

To alleviate the fairness issues, we proposed two experimental modifications to the general algorithm. The first, seeding, makes it extremely unlikely that the bad fairness case will be reconstructed. The second, Low Usage Credit, allows for violation of the protocol in a controlled manner, however it loses the minimum rate guarantee afforded by the Early Start algorithms. In our simulation experiments, both modifications improved the fairness, but the best (max-min fair) results came when combining both Low Usage Credit and seeding. We caution that the experiments only consider steady rate demands.

Of the presented algorithms, we recommend the Early Start Unrestricted or Early Start Steady algorithms, depending on whether steady output rate is preferred, as well as seeding. We recommend setting the global bucket threshold $G = (n-1)LT$ since this is the smallest value that guarantees full utilization; larger values increase the burst size and error. This makes the burst under $3nLT$. Then the parameter LT can be set smaller for smaller error, or larger for less frequent communication (reports) between policers. While we can not say whether our algorithms are better than predictive solutions in a practical setting, many of our theorems (starvation, burst size, minimum rate) are guaranteed under any arbitrary traffic conditions, and not just under expected conditions.

5.2 Future Work

We have focused on the theoretical guarantees we can get under certain arbitrarily bad scenarios, however the steady demand cases may not reflect real world usage. It would be interesting to see how well the algorithms perform under a Poisson model of packet arrivals, or dynamic traffic demands following the TCP protocol. It would be nice to see if the recommended Early Start algorithms with seeding are sufficiently fair in practice. Also, the analysis of the Low Usage Credit algorithm is relatively difficult compared to the others. We still do not know the ideal value of $CR_{\max}$, the maximum credit a policer may accumulate, to give adequate fairness. Finding this would give a better idea of the maximum error we can expect from this algorithm.

Additionally, we have only looked at local policer algorithms for improving fairness. It has been suggested to replace the logic at the master with a different, fair packet scheduling algorithm, since the role of the master is essentially to schedule when reports from each policer may arrive. However, our observations — that a constant traffic flow at policers can translate to uneven rates of reports arriving at

the master — show that fairness of reports does not necessarily mean fairness at the packet level. We suspect that better traffic fairness requires a longer "memory" at the master, which may impact full utilization. This certainly warrants further study.

References

[1] T. Bonald, L. Massoulié, A. Proutière, and J. Virtamo. A queueing analysis of max-min fairness, proportional fairness and balanced fairness. *Queueing Syst. Theory Appl.*, 53(1-2):65–84, 2006. 6

[2] M.J.C. Buchli, D. De Vleeschauwer, J. Janssen, and G.H. Petit. Policing aggregates of voice traffic with the token bucket algorithm. In *Communications, 2002. ICC 2002. IEEE International Conference on*, volume 4, pages 2547–2551 vol.4, 2002. 8

[3] M. Butto, E. Cavallero, and A. Tonietti. Effectiveness of the 'leaky bucket' policing mechanism in ATM networks. *Selected Areas in Communications, IEEE Journal on*, 9(3):335–342, Apr 1991. 8, 10

[4] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. Program. Lang. Syst.*, 11(1):90–114, 1989. 30

[5] R.J. Gibbens and F.P. Kelly. Distributed connection acceptance control for a connectionless network. *16th International Teletraffic Congress in Edinburgh*, Jun 1999. 8, 10

[6] J. Heinanen and R. Guerin. A single rate three color marker. Network Working Group, RFC 2697, Sep 1999. 4

[7] J. Heinanen and R. Guerin. A two rate three color marker. Network Working Group, RFC 2698, Sep 1999. 4

[8] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY, USA, 1988. ACM. 6, 16

[9] Raj Jain, Dah-Ming Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report DEC-TR-301, DEC, Sep 1984. 6

[10] T. Johnson. Designing a distributed queue. In *Parallel and Distributed Processing. Proceedings. Seventh IEEE Symposium on*, pages 304–311, Oct 1995. 30

[11] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49:237–252(16), Mar 1998. 1

[12] Frank Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8:33–37, 1997. 1, 6

[13] Srisankar S. Kunniyur and R. Srikant. An adaptive virtual queue (AVQ) algorithm for active queue management. *IEEE/ACM Trans. Netw.*, 12(2):286–299, 2004.

[14] P.P. Mishra. Effect of leaky bucket policing on tcp over atm performance. In *Communications, 1996. ICC 96, Conference Record, Converging Technologies for Tomorrow's Applications. 1996 IEEE International Conference on*, volume 3, pages 1700–1706 vol.3, Jun 1996.

[15] Ketan A. Padwekar. System and method for performing distributed policing. Patent, Oct 2006. US 2006/0221819 A1. 11, 12, 13, 31, 33, 36

[16] C. Partridge. A proposed flow specification. Network Working Group, RFC 1363, Sep 1992. 8, 9

[17] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 337–348, New York, NY, USA, 2007. ACM. 1, 5, 7, 15, 16, 17, 18, 23

[18] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, 1996. 20

[19] Rade Stanojević and Robert Shorten. Fully decentralized emulation of best-effort and processor sharing queues. In *SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 383–394, New York, NY, USA, 2008. ACM. 5, 7, 10, 17, 18, 20, 23, 54

[20] Puqi Perry Tang and T.-Y.C. Tai. Network traffic characterization using token bucket model. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 51–62, Mar 1999. 8, 9

[21] J.S. Turner. New directions in communications (or which way to the information age?). *Communications Magazine, IEEE*, 24(10), Oct 1986. 8

[22] Haïkel Yaïche, Ravi R. Mazumdar, and Catherine Rosenberg. A game theoretic framework for bandwidth allocation and pricing in broadband networks. *IEEE/ACM Trans. Netw.*, 8(5):667–678, Oct 2000. 1

[23] N. Yin and M.G. Hluchyj. Analysis of the leaky bucket algorithm for on-off data sources. In *Global Telecommunications Conference, 1991. GLOBECOM '91. 'Countdown to the New Millennium.*, volume 1, pages 254–260, Dec 1991. 9