Multidimensional Fair-share Rate Limiting in BPF

Jonas Otten
jonas@cloudflare.com

August 28, 2020

***Abstract* - As UDP does not have flood attack protections such as SYN cookies and sockets often have a single receive queue which can block in a flood scenario the whole application, we developed a novel fair-share ratelimiter *rakelimit* in unprivileged BPF, designed for a UDP reverse proxy, that is capable of applying rate limits to specific traffic streams while minimizing the impact on others. To achieve this, we base our work on Hierarchical Heavy Hitters, which proposes a method to group elements on attributes such as source and destination IP address, and we are able to substantially simplify the algorithm for our rate-limiting use case to allow for an implementation in BPF. We further extend the concept of a hierarchy from IPs addresses to ports, providing us with precise rate limits based on the 4-tuple. Our approach is capable of rate-limiting floods originating from single addresses, subnets but also reflection attacks, and applies limits as specific as possible. To verify the performance, we evaluated the approach against different simulated scenarios. This project's outcome is a single *Go*-library that can be activated on any UDP socket and provides flood protection out of the box.**

1 Introduction

As Internet usage surges [1] dependencies on its availability increase since social interactions have been moved to video calls, and a large number of people move their workplace to a work-from-home setup. To ensure availability at any given point in time, server applications have to be protected from floods of packets rendering a service unusable. Several services on the Internet rely on the *UDP* protocol, which popularity will likely surge in the future with the broad adoption of *QUIC* and *HTTP/3*. While the *TCP* protocol includes different mechanisms to provide a basic level of protection against floods, such as *syncookies* [2], but the Linux kernel also provides mechanisms such as conntrack as part of the netfilter subsystem [3]. While these mitigations are available for *TCP*, the connectionless protocol *UDP* does not have such protections. This work proposes a new fair-share rate-limiting algorithm based on *Hierarchical Heavy Hitters* [4] which goal it is to detect common floods scenarios including attackers from a single address, a single subnet and a reflection attack originating from a single source port. The algorithm should be simple enough to be implemented in a BPF socketfilter, ensuring we can rely on the small overhead possible with BPF without exposing a larger security surface. We aim for a simple and easy to use library which is capable to prevent floods out-of-the-box of various types with a minimal configuration required.

2 Design

2.1 Estimating Counts

The main objective is to detect traffic streams, which take a large fraction of the overall throughput. Once a stream is identified that exceeds the limit, only a subset of packets of that stream should be allowed to pass to comply with the threshold. A rate is defined

in *packets per second* (pps) This closely relates to the problem of finding *Heavy Hitters*: determining items that take a large fraction of a stream. To identify those, simply keeping a rate per each 4-tuple is not scalable as the number of combinations would exceed memory limits. Instead, probabilistic algorithms can be used, such as a SpaceSaving algorithm [5] or a CountMin sketch [6], which aim to provide an estimate of an element in an infinite stream with constant memory requirements. As a CountMin sketch does not require a lot of complex code and stores small amounts of data due to the hashing process, we decided to swap it in for the SpaceSaving algorithm.

A CountMin sketch [6] is a probabilistic datastructure capable of providing an estimated count of an arbitrary element with constant space requirements. It can update a count by passing an element through different hash functions h and increasing its respective items in the array of size w. When estimating a count, the same procedure is repeated, but counts are stored for each hash function, and the minimum count is the estimate for the element. It expects two parameters (e, δ) provide with the probability $p_{correct}$ an estimate $\hat{x}$

$$p_{correct} \geq 1 - \delta \tag{1}$$

$$\hat{x} \leq x + \epsilon \|x\| \tag{2}$$

These parameters determine the width w and the height h of the internal two-dimensional array.

$$w = \lceil e/\epsilon \rceil \tag{3}$$

$$d = \lceil ln1/\delta \rceil \tag{4}$$

Rakelimit aims for an error of 1% with the probability 0.01, resulting in $w = 273$ and $h = 5$.

$$w = \lceil e/0.01 \rceil = 271.83 \approx 272 \tag{5}$$

$$h = \lceil ln1/0.01 \rceil = 4.61 \approx 5 \tag{6}$$

To avoid bias in the hashing process, the width is aligned to a power of 2, for which we accept a small increase in the error, thus $w = 256$. This results in an error of approx. 0.011.

$$\lceil \epsilon \rceil = e/256 \approx 0.011 \tag{7}$$

These parameters may change in the future due to further optimizations.

2.1.1 Estimating Rates

Instead of counts, *rakelimit* stores rates in the CountMin sketch. A rate r is defined as

$$r = \frac{1}{t_{now} - t_{prev}} \tag{8}$$

where t is a time in seconds.

Rate limiting based on the most recent rate is not precise since it would only capture the rate between two packets, thus only being based on a varying timespan. Instead *EWMA* [7] is used. *EWMA* essentially takes an old rate and a rate that was just measured and combines these two to generate a more accurate estimate. This results in rates being estimated based on a time-based sliding window instead of the timespan between the last two packets.

$$r_{current} = \frac{1}{dur_{seconds}} \tag{9}$$

$$r_{estimate} = a \times r_{current} + (1 - a) \times r_{old} \tag{10}$$

a determines the impact of the rate just measured on the estimated rate. To determine this factor, the timestamp of the previously measured rate is used t_{prev}.

The older the previous timestamp t_{prev}, the smaller the impact of the previous rate should be. Thus

$$a = \begin{cases} \frac{t_{now} - t_{prev}}{window} & \text{if } t_{now} - t_{prev} \leq window \\ 0 & otherwise \end{cases}$$

The parameter *window* determines how long an element is considered relevant/accurate.

rakelimit as of now uses a window of 1*second* which has been chosen as a first naive parameter, and keeps the same mechanism as previously described to store these in a *CountMin* sketch. The *window* parameter will be changed once more knowledge around its impact is collected.

2.2 Finding Traffic Streams

The main contribution of this work is the grouping of packets, thus finding traffic streams. The general idea is to take a 4 tuple and generalise it in different ways along the four dimensions. When this is done multiple times extracting different parts, a rate limit can be applied to each of these generalisations. For example an address port combination *127.0.0.1:1234* can be extracted into *127.0.0.*:1234*, *127.0.0.1:**, and so on.

An incoming packet would update the rates for all of these generalisations, thus allowing to detect floods from a single address, a whole subnet, and others. This idea is based on *Hierarchical Heavy Hitters* (in particular [8]), which is used to determine *Heavy Hitters* in datastreams. It relies on the implicit hierarchy by IP addresses, as shown in Figure 1.

We extend this structure to be applicable to ports, which are either *specified* or since they are fully arbitrary a *wildcard*.

Due to our application of rate limiting we are able to substantially simplify the algorithm proposed in [8], as we aim for no Heavy Hitters at any point in time, resulting in Algorithm 1

This algorithm takes an element and determines if it exceeds any rate limit, and if so prints the rate. A level is defined as the amount of generalisations that has been applied to an element, as illustrated in the following example:

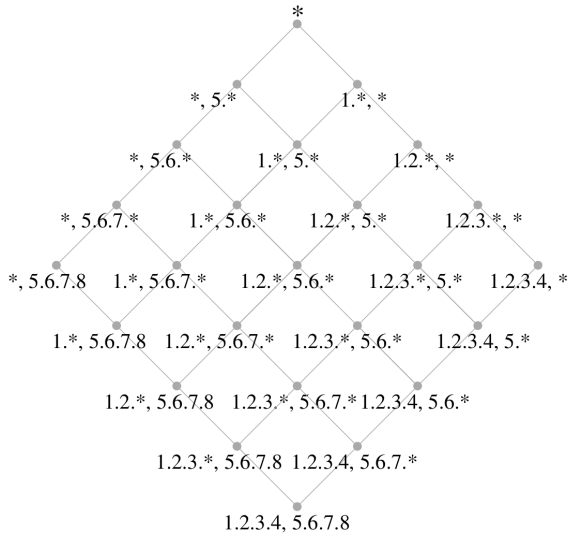

Figure 1: A two-dimensional lattice for IP addresses, from [4, p.7]

Level 0: 127.0.0.1:5432

Level 1: 127.0.0.1:*, 127.0.0.*:5421

Level 2: 127.0.0.*:*

While *rakelimit* considers not just the source address and port but the 4 tuple, the logic is equivalent, and we consider the following 12 combinations with IPv4 packets:

Source Address	/32, /24, /0
Source Port	Specified, Wildcard
Destination Address	/32
Destination Port	Specified, Wildcard

An equivalent IPv6 setting is defined as

Source Address	/64, /48, /0
Source Port	Specified, Wildcard
Destination Address	/128
Destination Port	Specified, Wildcard

Algorithm 1: Hierarchical Heavy Hitters for Rate Limiting

```
φ: rate_limit;
for level l = 0, l < L, l = l + 1 do
    foreach item p at level l do
        Let n be the lattice node that p
         belongs to
        // get last rate;
        rate_prev = GetEstimateCM(CM(n), p);
        rate_current = CalculateRate(t_now,
         t_prev, rate_prev);
        if rate_current ≥ φ then
            print(p, rate_current);
            return // important, as packet has
             to be dropped from this point on;
```

2.3 The Algorithm

The rate-limiting algorithm consists of two parts. First, on each incoming element, different generalisations are computed, and rates for the respective elements are updated. If a rate exceeds the rate limit in any of the updated generalisations, it gets dropped, otherwise, it is allowed to pass.

While this seems obvious, there are some crucial details to consider.

We want rate limits to be applied as specific as possible, but as loose as necessary. That means that if a flood originates from a fully specific four tuple, only this tuple should be rate limited. No other packets should be impacted. If the flood originates from a whole /24 subnet, the whole subnet should be rate limited. This means that a rate limited packet is never counted against more general rate limits, only against more specific ones.

A first idea to use this logic would be to do the rate limiting in two steps. First, each element is passed to the nodes and a rate is determined. If all rates are below the rate limit, the packet is allowed to pass and is updated in all of the nodes. Otherwise it will be simply dropped. This works but has the drawback that it does not converge to specific rate limits, it will stay where the rate limit first happens.

Instead, we propose to combine checking and adding elements. Generalisations of an element are added to the respective *CountMin* sketch, and after each level the maximum rate of that level is compared to the rate limit. If the max rate is below the rate limit, the next level will be checked. Otherwise, the elements is considered to be part of a flood, and will not be added to higher (more generic) levels.

Packets that are part of are either allowed to pass immediately with a probability p_{pass}, and gets dropped otherwise.

$$p_{pass} = \frac{\text{rate limit}}{\text{max rate}} \tag{11}$$

The passing probability ensures that flood streams get not fully blocked but can still pass within the limit.

The whole algorithm is shown in Figure 9

A rate limit is capable of converging towards more specific rate limits, but due to its structure also effectively isolates from other traffic. This ensures that the impact on legitimate traffic is minimised. The proposed algorithm is shown in Algorithm 2.

Algorithm 2: Hierarchical Heavy Hitters for Rate Limiting

```
φ: rate_limit;
max_rate = 0;
for level l = 0, l < L, l = l + 1 do
    foreach item p at level l do
        Let n be the lattice node that p
         belongs to
        // get last rate;
        rate_prev = GetEstimateCM(CM(n), p);
        rate_current = CalculateRate(t_now,
         t_prev, rate_prev);
        if rate_current > max_rate then
            max_rate = rate_current;
    if max_rate > φ then
        if randFloat() ≤ rate_limit/max_rate
         then
            return PASS;
        return DROP;
```

3 Implementation

In order to provide a low-overhead solution, the algorithm is implemented in *BPF (Berkeley Packet Filter)*. As the goal of this project is to provide a simple-to-use *Go* library that can be activated on any socket, it is implemented as a socketfilter to allow an impact on other applications, as well as a minimal security impact as it does not require root privileges. The CountMin sketch is stored in a *BPF* map. *rakelimit* uses *fasthash* [9], since it is fast, small and accepted by the BPF verifier.

To implement a CountMin sketch in *BPF* a hash-function has to be determined, and the counts have to be persisted across packets. As we require multiple CountMin sketches, which will be covered later, a *BPF* array is used. The code we use is shown in Snippet 3.

```
struct cm_value {
  fpoint value;
  __u64 ts;
} __attribute__((packed));

struct countmin {
  struct cm_value values[HASHFN_N][COLUMNS];
} __attribute__((packed));

```

CountMin sketch in BPF

In *rakelimit*, querying for counts to determine what the current rates are and potentially adding a packet happens for each packet, and due to the algorithm can be merged together. The implementation which stores both a rate and a timestamp is shown in Snippet 3.

We pass various parameters such as the current timestamp, used to calculate a rate, the CountMin struct, and a struct holding the element we want to query and update.

Then we iterate through all hash functions and generate a hash using fasthash. The index determines the position of the item in the respective array of the hash function. This item holds both a previous rate and a timestamp, which we both update and then check if the newly determined rate is smaller than the minimum, and if so, we update the minimum. After all iterations we return the minimum rate found.

We use *EWMA* to calculate rates based in *packets per second (pps)*, as this is a natural format to represent rates in. To implement it a fixed-point representation is required in order to represent the factor a. To avoid further complexity, we ensured that no signed fixed-points are used. The code to estimate a new rate based on an old rate and a rate that was just measured is shown in Snippet 3

The code converts the numbers a few times back and forth between integers and fixed-points. This is due to the fact that we want to avoid over- and underflows as much as possible. While adding and subtracting of two fixed-points works without any issues as long as the result fits into the fixed-point precision, it does not apply for multiplications and divisions. When multiplying or dividing with two fixed-points, the factor introduced by the fixed-point representation has to be discounted (multiplication) or accounted (division). This intuitively would result in the following formula for multiplication:

$$result_{mul} = fp_a * fp_b/(2 << 32) \tag{12}$$

And in a similiar situation for division:

$$result_{div} = fp_a/fp_b * (2 << 32) \tag{13}$$

While this may work out for certain use-cases, the intermediate results of the operations may over or underflow the 64 bits available, even though the actual result would fit into the 64-bit fixed-point representation.

To avoid such a scenario, we only use multiplications and divisions with mixed representations,

```
static __u64 FORCE_INLINE add_to_cm(__u64 ts, struct countmin *cm,
struct packet_element *element)
{
fpoint min = -1;

#pragma clang loop unroll(full)
for (int i = 0; i < HASHFN_N; i++) {
  __u32 target_idx = fasthash64(element, sizeof(struct packet_element), i) & (COLUMNS-1);

    struct cm_value *value = &cm->values[i][target_idx];
    value->value = estimate_avg_rate(value->value, ts - value->ts);
    value->ts = ts;

    if (value->value < min) {
      min = value->value;
    }
}
return min;
}

```

Querying and adding to a CountMin sketch in BPF

```

__u64 rate_current = 1000000000ull / dur;
// last timestamp > current timestamp + duration? use new rate
if (dur >= WINDOW) {
    return to_fixed_point(rate_current);
}

fpoint a = to_fixed_point(dur) / WINDOW;

fpoint new_rate = old_rate;
if (old_rate > to_fixed_point(rate_current)) {
  new_rate -= a * to_int(old_rate - to_fixed_point(rate_current));
} else {
    new_rate += a * to_int(to_fixed_point(rate_current) - old_rate);
}
return new_rate;

```

Calculating rates in BPF

meaning that one argument is a fixed point while the other argument is an integer.

It is important to note that the result of such an operation is again a fixed point, as the multiplication of a fixed-point with the integer 5 does not require any further discounting. This ensures no over- or underflows happen in the intermediate results, and resolves the previously mentioned issue. It is important to note that while it does not matter which factor is in fixed-point representation, when dividing the divisor has to be an integer and the dividend has to be in fixed-point representation.

As we store rates in *packets per second (pps)* and this component is aimed to mitigate floods and therefore large rates, there is no need to determine the rate more precise than an integer. This allows us to convert these parameters back and forth between types without degrading the quality of it.

While the limitations of *BPF* have required a few workarounds, such as missing bpf-to-bpf calls or the missing context fields in the *skb* struct, the whole algorithm could be implemented without any further issues.

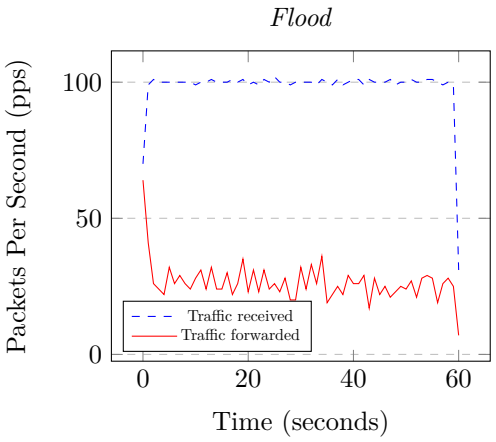

Figure 2: A flood from a single address and a single port. The rate limiter successfully limits it to 25 *packets per second*

4 Results

To ensure the quality of the algorithm and the implementation, different simulations have been run to examine how well the rate limiter can limit floods and isolate their impacts. In the following scenarios, a *flood* stream and other traffic streams happening at the same time will be examined.

4.1 Scenario 1: Single address & port

A flood is generated with 100 *packets per second* from a single address and single port *127.0.0.1:80*. Another stream falling in the same subnet *127.0.0.1:80* at the same time with 5 *packets per second* is generated to examine the impact on legitimate traffic. The results are shown in Figure 2 and Figure 3, confirming that the flood gets successfully rate-limited without impacting traffic, even if it falls within the same subnet.

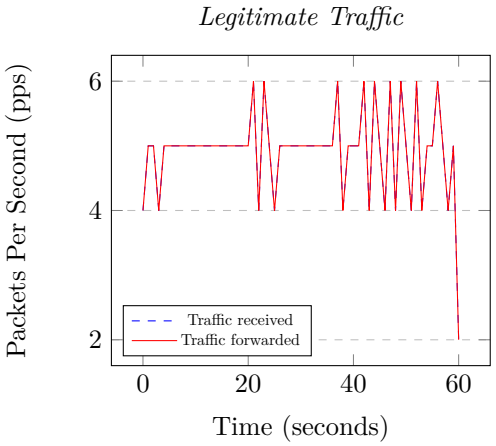

Figure 3: Legitimate traffic from the same subnet as the flood. The rate limiter is capable of isolating the flood completely, resulting in no impacts for other traffic.

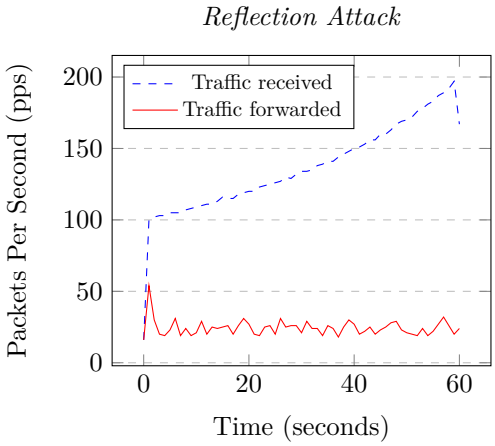

Figure 4: A reflection attack from random addresses using a single port.

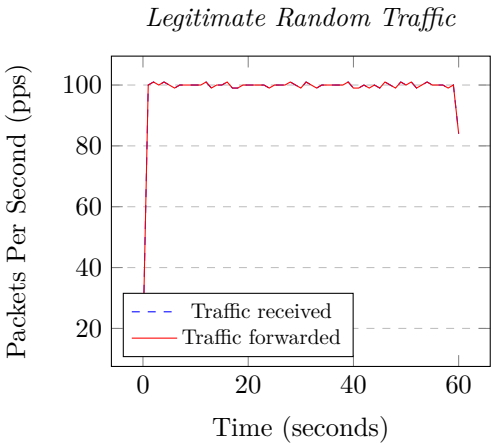

Figure 5: Legitimate from packets with a fully random 4 tuple. As the rate limiter only limits a singke soyrce port for a fully specified destination the majority of traffic is unimpacted.

4.2 Scenario 2: Reflection Attack

In this scenario, a reflection attack with 100 *packets per second* is simulated, with traffic being send from random IP addresses but a single port. To confirm the impact on other parties, a traffic stream consisting of packets with a fully random 4 tuple will be generated to ensure the rate limiter does not just limit the overall throughput. The results are shown in Figure 4 and Figure 5.

This confirms that even if packets have to be aggregated multiple times (fully specified address → /24 subnet → full wildcard /0) the rate limiter impacts as little other traffic as possible.

4.3 Scenario 3: High throughput & limit

To ensure the rate limiter is capable of dealing with large floods, a third scenario simulates 5 seconds of flood traffic at a rate of 1,000,000 *packets per second*, with a rate limit of 250,000 *packets per second*. The flood traffic is shown in Figure 6.

The rate limiter behaves exactly as previously, just on a larger scale. The initial spike persists, but as previously noted, can be reduced by reducing the window size used.

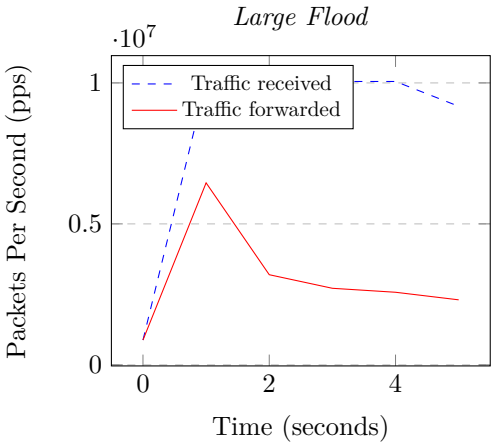

Figure 6: A large flood with a large rate limit, showing the same behaviour as in smaller attacks. The spike seems to persist longer, which is due to the smaller duration (5 seconds instead of 60)

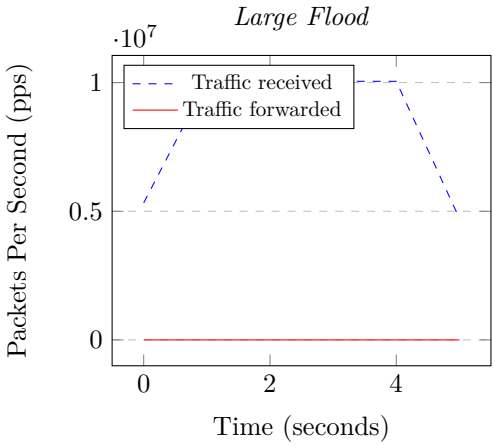

Figure 7: A large flood with a small rate limit, highlighting that the spike is independent of the input rate but proportional to the rate limit.

Time (second)	Received	Forwarded
0	5425698	361
1	10050250	41
2	10050182	34
3	10050446	39
4	10050261	23
5	4724513	13

Figure 8: Received and forwarded packets counts of a large flood with a small rate limit

4.4 Scenario 4: High throughput small limit

The last scenario uses the same flood as in *Scenario 3*, simulated for five seconds with 1,000,000 *packets per second*, but uses a comparable small rate limit 25 *packets per second*. The flood traffic is shown in Figure 7.

The scenario highlights that the initial spike is independent of the input rate but proportional to the rate limit. As it is impossible to visually extract further information, the raw data is listed in Table 8.

The forwarded packets initially exceed the rate limit, but converge in the subsequent seconds towards the set rate limit. This confirms that there are no significant imprecisions that could impact the quality of the rate limiter.

5 Conclusion

The results of *rakelimit* over are very satisfactory. The algorithm is capable of detecting and isolating traffic streams based on attributes of the 4 tuple. The impact on other traffic streams is kept at a minimum, ensuring floods do not degrade services. It has been confirmed that a *BPF* socketfilter is capable of handling such a complex scenario, ensuring that a minimal security surface is exposed while gaining the low overhead of the kernel space. Future work includes evaluating the algorithm with first checking in every node if it exceeds the rate limit, and only if it doesn't add the packet to all generalisations of the level. This ensures that floods do not just get rate limited more specific, but also more generic in case different floods start at different times. The algorithm is successfully implemented in *BPF* and will be further optimised before it will be open-sourced in September on Github under *cloudflare/rakelimit*.

References

[1] M. Sweney, "Vodafone reports 50% rise in internet use as more people work from home," *The Guardian*, 2020.

[2] P. McManus, "Improving syncookies," *lwn.net*, 2008.

[3] M. Majkowski, "Conntrack tales - one thousand and one flows," *Cloudflare Blog*, 2020.

[4] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in streaming data,"

[5] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proceedings of the 10th international conference on Database Theory*, ICDT'05, Springer-Verlag.

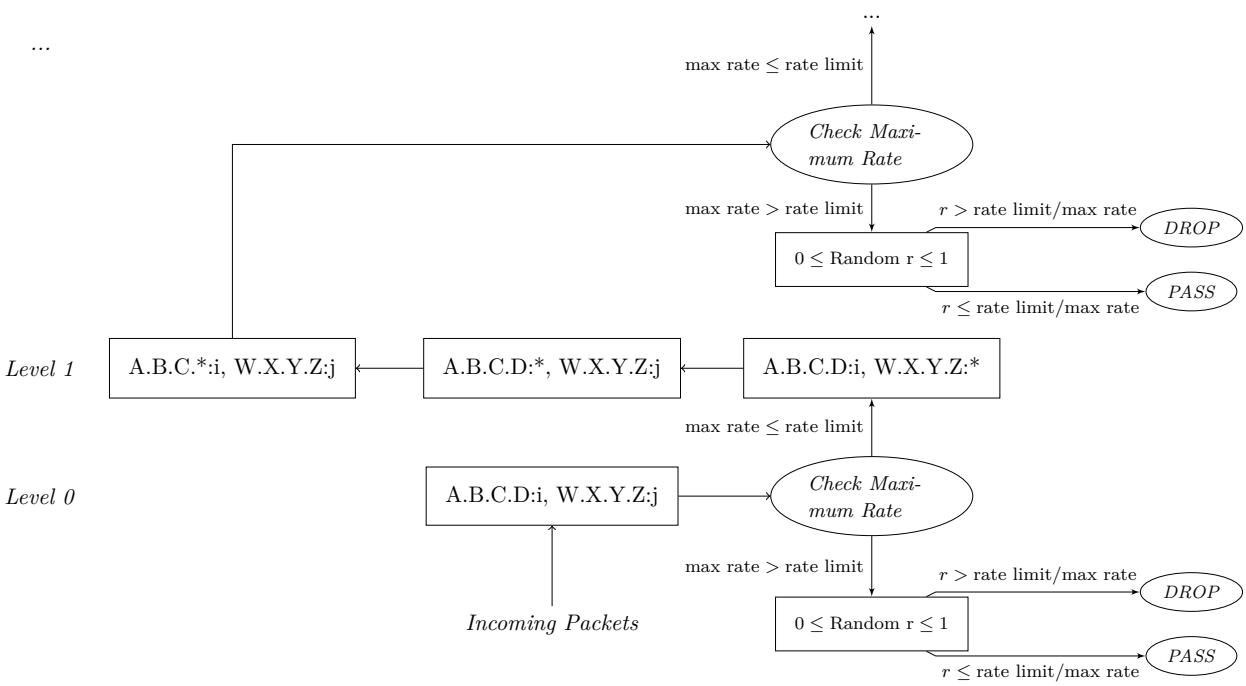

Figure 9: Processing of packets using the probabilistc method on a level-basis. Maxima are kept per level to decide if a packet should pass or not.

[6] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications,"

[7] "6.3.2.4. EWMA control charts,"

[8] M. Mitzenmacher, T. Steinke, and J. Thaler, "Hierarchical heavy hitters with the space saving algorithm," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, ALENEX '12, pp. 160–174, Society for Industrial and Applied Mathematics.

[9] eric, "ztanml/fast-hash." original-date: 2018-10-22T16:16:17Z.