Fair-share Rate limiting in BPF

Jonas Otten
Lorenz Bauer
Cloudflare
Outline

- Introduction
- Algorithm
- BPF Implementation
- Conclusion
Introduction - DNS & QUIC

DNS & QUIC services
  ○ E.g. 1.1.1.1

→ UDP services
Introduction - Problem

DNS & QUIC services
  ○ E.g. 1.1.1.1

What about Floods?
Introduction - Problem

DNS & QUIC services
  ○ E.g. 1.1.1.1

→ Flood & legitimate traffic on same socket
→ Queues will overflow, traffic be dropped
Introduction - Problem

● TCP: SYN-cookies
→ no similar context for UDP

How can we protect UDP sockets?
Solution: Rate Limiting
Solution: Rate limiting

Idea
1. detect the traffic stream that’s flooding
2. rate limit
What is a “stream”? 

- Need to group packets based on some criterion
First Idea: match on part of 4-tuple

- Source Port
- Source Address
- Destination Port
- Destination Address

→ rate limit each of those
First Idea: match on part of 4-tuple

Source Port | Source Address /32 | Destination Address /32 | Destination Port
First Idea - Algorithm

Source Port → Source Address /32 → Destination Address /32 → Destination Port
First Idea: limitations

● Susceptible to DoS
  ○ E.g. Source Ports can be flooded easily
Idea: Evaluate Attributes simultaneously

● Based on *Hierarchical Heavy Hitters*
  ○ Simplifications due to rate limiting scenario

● Combine all attributes
  ○ → Lattice
Grouping packets

Packet A
10.11.12.100

Packet B
10.11.12.231

Packet C
192.168.0.20
Grouping packets

- Subnet 10.11.12.0/24
  - Packet A 10.11.12.100
  - Packet B 10.11.12.231
  - Packet C 192.168.0.20

Subnet 0.0.0.0/0
Incoming packet from 10.11.12.231

PASS

Subnet 0.0.0.0/0

+1pps

Subnet 10.11.12.0/24

Packet A
10.11.12.100

Packet B
10.11.12.231

Packet C
192.168.0.20

+1pps
Incoming packet from 10.11.12.231
Hierarchy of a single attribute

/0

*.*.*.*: 5pps

/24

192.168.0.*: 1pps
10.11.12.*: 4pps

/32

192.168.0.20: 1pps
10.11.12.100: 1pps
10.11.12.231: 3pps
Multi-attribute Example

Level 0

127.0.0.1:80

Level 1

127.0.0.*:80

Level 2

127.0.*.*:80

127.0.0.*:*
Multi-attribute: Extendable to more attributes
BPF Implementation
Unprivileged socketfilter - Why?

1. Minimise required privileges
2. Minimise blast radius
3. Only Service is aware of its appropriate rate limit
Ingredients

- Determine rates of any element
- We need a sliding window over packet rates
CountMin Sketch: estimate counts

- Probabilistic data structure
- Estimate counts for any element
- Constant Memory Requirements
CountMin Sketch: estimate counts

<table>
<thead>
<tr>
<th>hashfn</th>
<th>3</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashfn</td>
<td>10</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>hashfn</td>
<td>4</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>
CountMin Sketch: Updates

hashfn_1

3 + 1  7  9

hashfn_2

10  2  12 + 1

hashfn_3

4 + 1  3  20
## CountMin Sketch: Queries

<table>
<thead>
<tr>
<th>hashfn_1</th>
<th>4</th>
<th>7</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashfn_2</td>
<td>10</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>hashfn_3</td>
<td>5</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>

Estimate Count: 2
CountMin Sketch: estimate counts

- Probabilistic data structure
- Estimate counts for any element
- Constant Memory Requirements

→ works great with BPF array
→ fasthash to hash elements
CountMin Sketch: estimate counts

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

struct bpf_map_def SEC("maps") countmin = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(__u32),
    .value_size = sizeof(struct countmin),
    .max_entries = NODES,
};
```
CountMin Sketch: estimate counts

```c
// add element and determine count
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;
}
```
Determining Rates

Naive:
rate = 1/duration since last packet

→ Loose all context before
→ Instead: Exponentially Weighted Moving Average (EWMA)
EWMA: discounting rates over time

Weight of Measurement

Time
EWMA: discounting rates over time

Previous rate, duration since last packet
→ instant_rate = rate since last packet

\[
\text{estimate\_rate\_pps}(\text{instant\_rate}, \text{old\_rate}) : \\
\quad \text{return } (1-a) \times \text{instant\_rate} + a \times \text{old\_rate}
\]
EWMA: discounting rates over time

→ 0 <= a <= 1
→ use fixed points for discounting

```
estimate_rate_pps(instant_rate, old_rate):
    return (1-a) * instant_rate + a * old_rate
```
EWMA: discounting rates over time

```c
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;
```
It works!

```
jonas@nebelhorn:~/git/rakelimit$ ls -lh src/rakelimit_bpfel.o
-rw-rw-r-- 1 jonas jonas 875K Aug 20 16:07 src/rakelimit_bpfel.o
jonas@nebelhorn:~/git/rakelimit$
```
Testing

● Go Implementation as reference
  ○ Validate the algorithm
  ○ Validate Results of BPF Implementation

● Created flood scenarios
  ○ Single address/port flood
  ○ /24 {single,random} port flood
  ○ Reflection attack
Scenarios - Single address & port
Scenarios - Single address & port

10.11.12.13:89

10.11.12.15:89
Scenarios - /24 address & single port
Scenarios - /24 address & single port

10.11.12.*:89

10.11.12.15:89
Scenarios - /24 address & single port

10.11.12.*:89

10.11.12.15:90
Limitations of unprivileged Socketfilters
Limitations unprivileged socketfilter

• Missing context fields

```c
// does not work
*port = skb->remote_port;
// instead
*port = (load_byte(skb, 0) << 8) | load_byte(skb, 1);
```
Limitations unprivileged socketfilter

- No bpf to bpf calls

```c
#define FORCE_INLINE inline __attribute__((__always_inline__))

static FORCE_INLINE int process_packet(__u64 ts, struct __sk_buff *skb) {
```
Limitations unprivileged socketfilter

- No bpf to bpf calls

```c
#define FORCE_INLINE inline __attribute__((__always_inline__))

static FORCE_INLINE int process_packet(__u64 ts, struct __sk_buff *skb) {

  ○ Forced inlines: 875k
  ○ No inlines: 81k
```
Limitations unprivileged socketfilter

- No floating points (BPF wide limitation)

```c
static __u64 FORCE_INLINE to_fixed_point(__u32 n)
{
    return ((__u64)n) << FRACTION_BITS;
}
```
Limitations unprivileged socketfilter

● No floating points (BPF wide limitation)
  ○ Instead: Fixed-points
  ○ Warning: had to spend lots of time to avoid precision problems
Conclusion

● Rate Limiter impacts minimum set of streams
● Rates can be estimated in BPF
  ○ EWMA
  ○ Fixed Points
● Limitations in unprivileged BPF
  ○ Can be worked around
Future

Continue optimisation within Cloudflare
Benchmarking
Publish Blog post
Thanks!

jonas@cloudflare.com
lmb@cloudflare.com