draft with speaker notes

DoS Protection in Knot Resolver

using multi-prefix query counting

Lukáš Ondráček (pictures, most of implementation), Vladimír Čunát (presenting) • October 12, 2024

CZ.NIC CZ DOMAIN REGISTRY

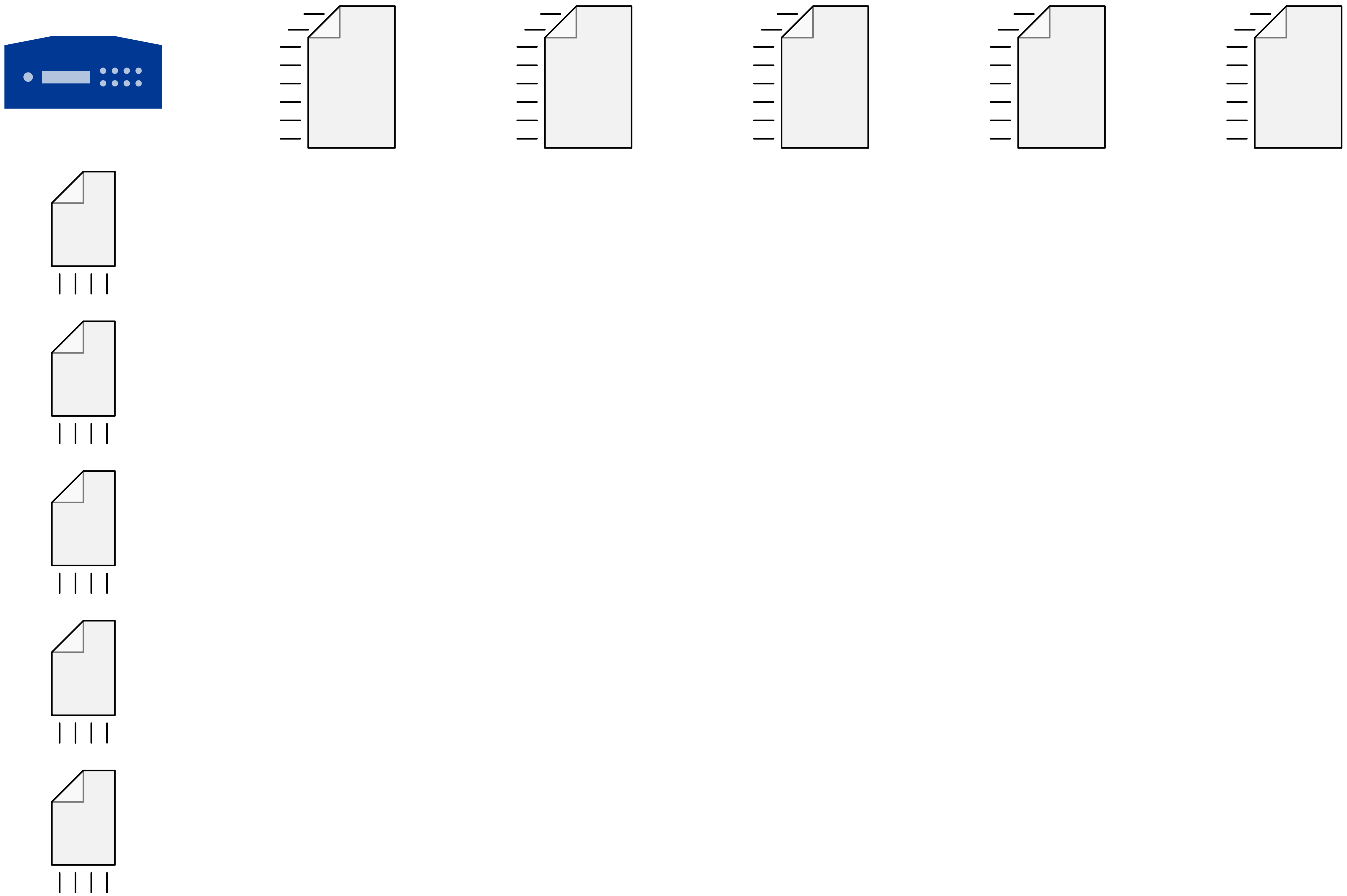
Introduction: DoS

- no rate limiting ability so far, adding now in 2024
- better *inside* resolver to understand DoT, DoH, and in future DoQ
- public resolvers are the main use case, e.g.:
  - cz.nic's ODVR: https://www.nic.cz/odvr/
  - DNS4EU instances (to become public in 2025)
- also mostly applied in authoritative Knot DNS >= 3.4

Overview

- still a common DoS technique
- attacks through some UDP servers where answers are bigger than queries, therefore amplifying the attacker's traffic

1. amplification att.
   - forged UDP source
   - answers >> queries

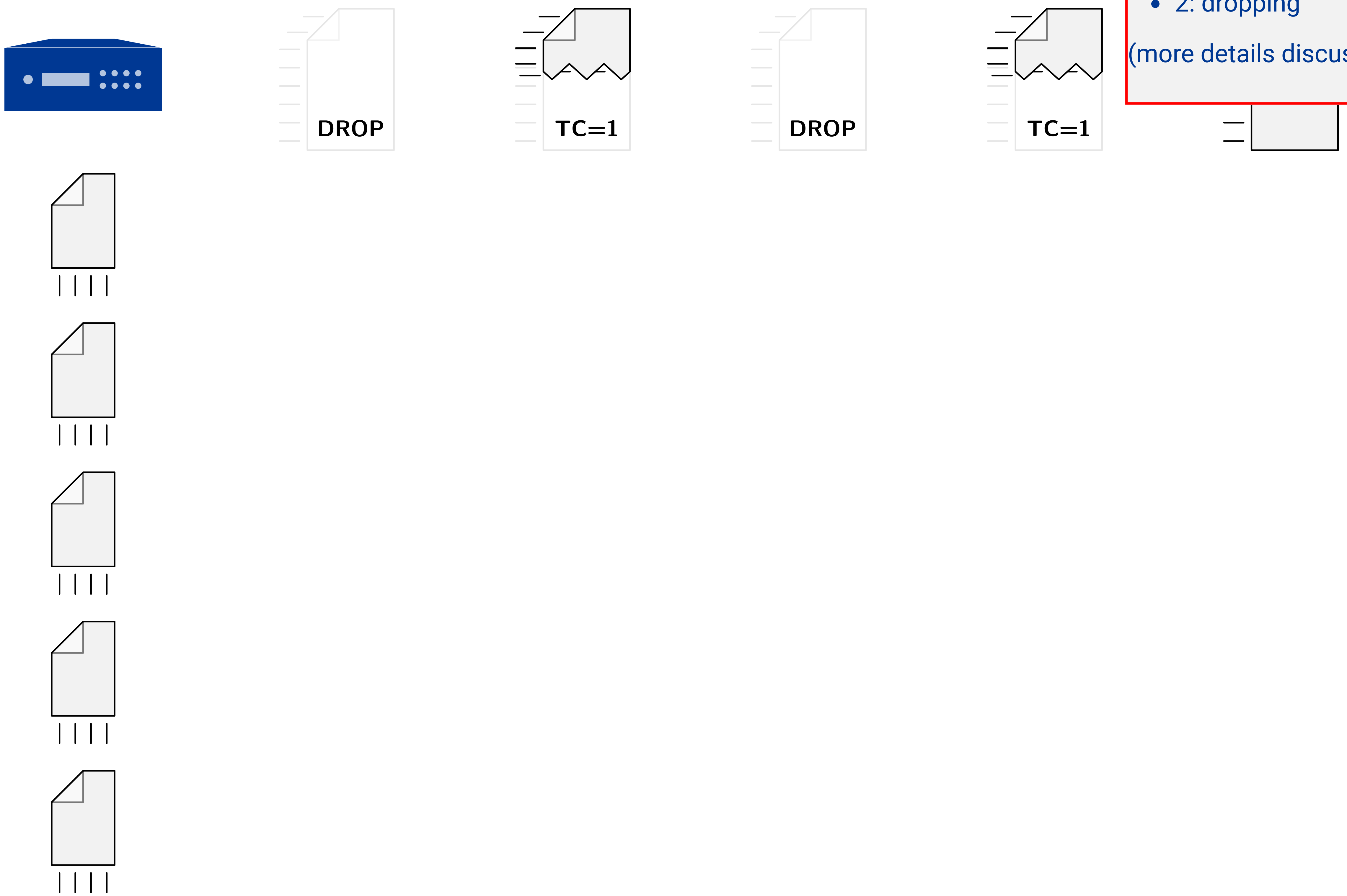


Overview

- Size of UDP replies was limited already, primarily to avoid issues with fragmentation.
- Now additionally: restrict response rate for each address/network, to protect *them*
- 1: truncation
  - same length of answer as its query, i.e. not amplifying
  - sane clients retry over TCP; there you can't forge source IP like that
- 2: dropping

(more details discussed later)

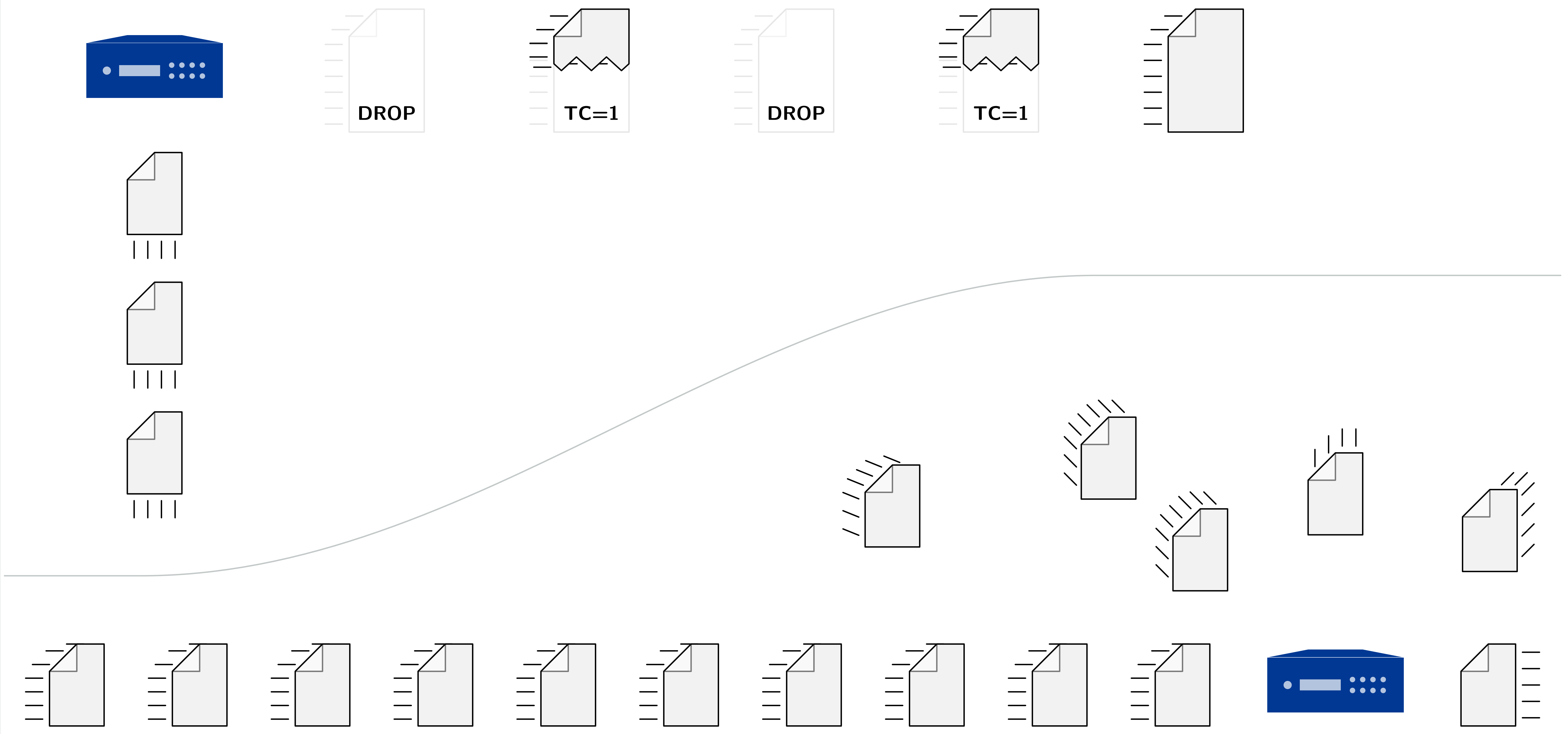
1. amplification att.
   - forged UDP source
   - answers >> queries
   - size limit (1232 B)
   - rate limiting
     - truncation
     - dropping



- Many requests from the same origin can exhaust cpu time…

Overview

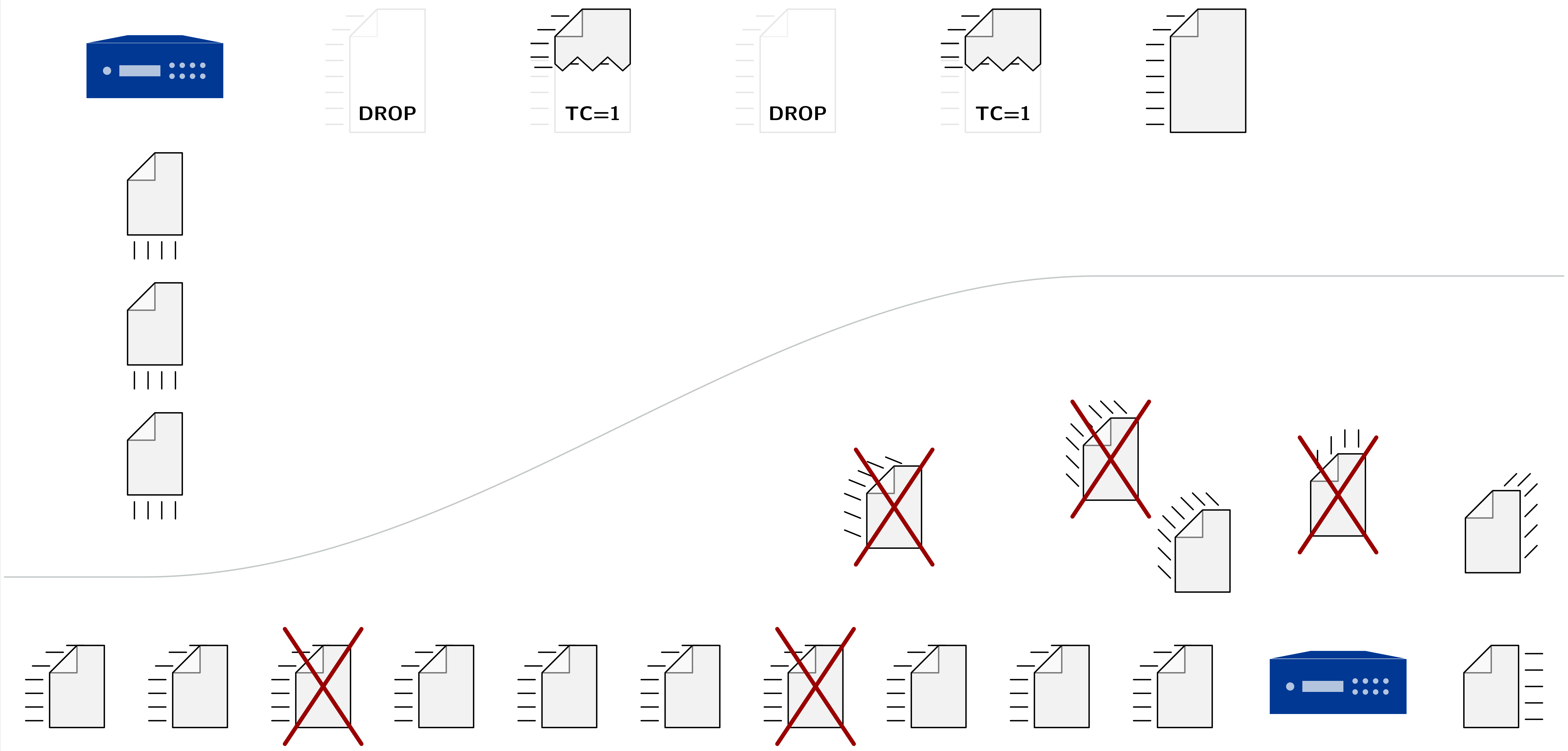
1. amplification att.
   - forged UDP source
   - answers >> queries
   - size limit (1232 B)
   - rate limiting
     - truncation
     - dropping
2. CPU overload



- …dropping some requests.

Overview

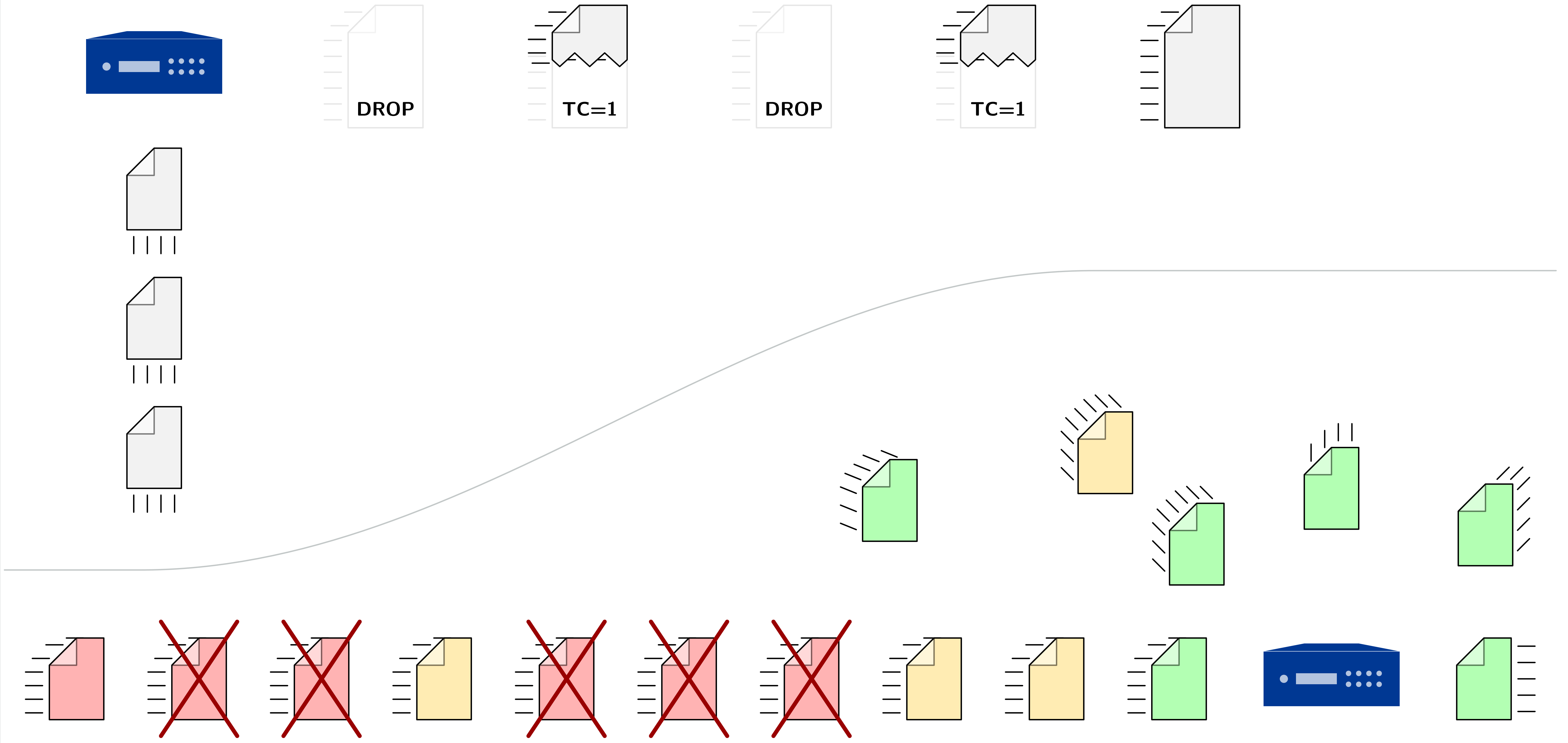
1. amplification att.
    - forged UDP source
    - answers >> queries
    - size limit (1232 B)
    - rate limiting
        - truncation
        - dropping
2. CPU overload



- Solution: defer requests from the origins using more cpu time in the past, so that users that do not overload the service shouldn't suffer.
- Non-UDP only, because on UDP the source IP could be faked.

Overview

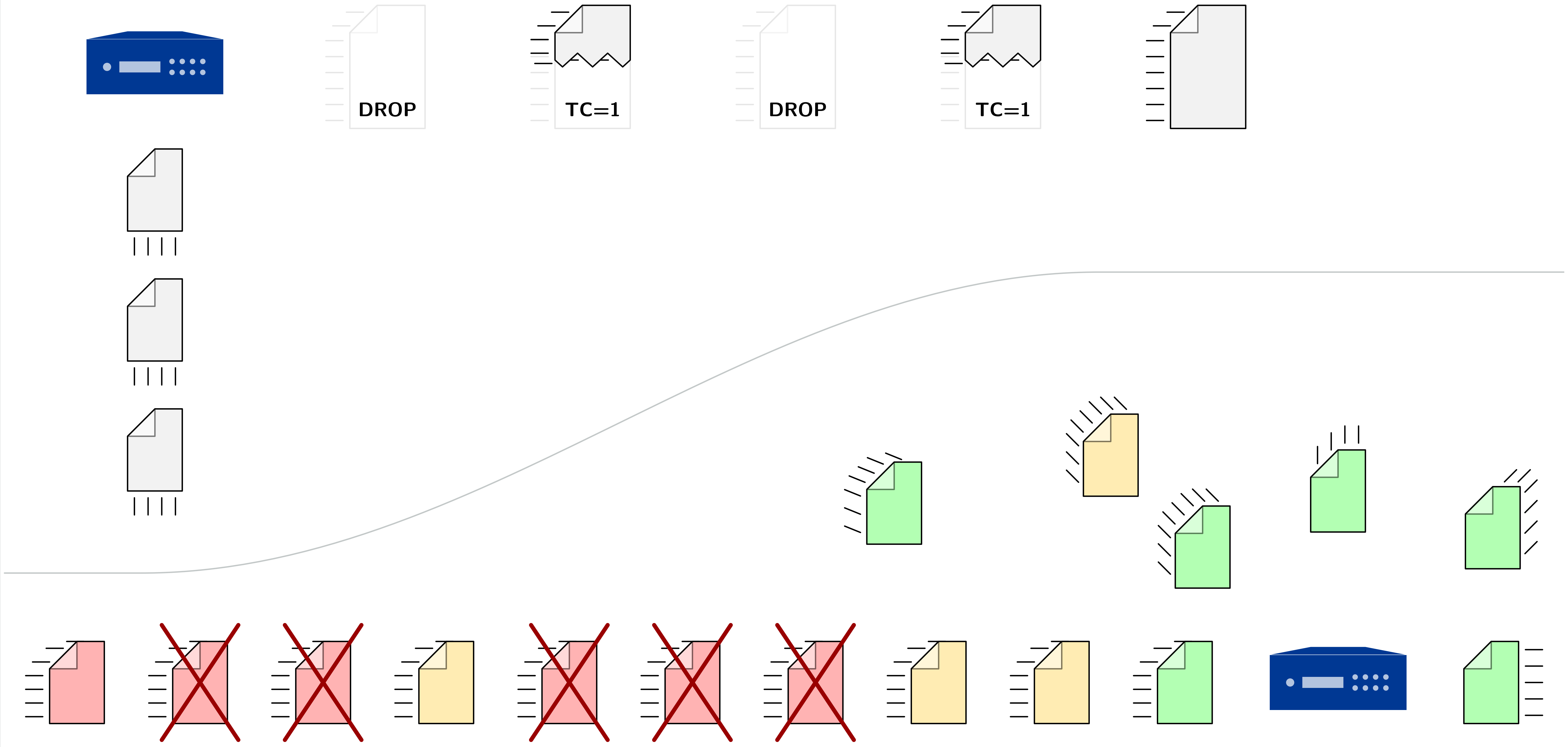
1. amplification att.
   - forged UDP source
   - answers >> queries
   - size limit (1232 B)
   - rate limiting
     - truncation
     - dropping
2. CPU overload
   - prioritization
     - all except plain UDP



Overview

- Outline:
  - limiting individual hosts,
  - extending to networks,
  - different limits for dropping and truncating,
  - prioritization,
  - ~~implementation details.~~

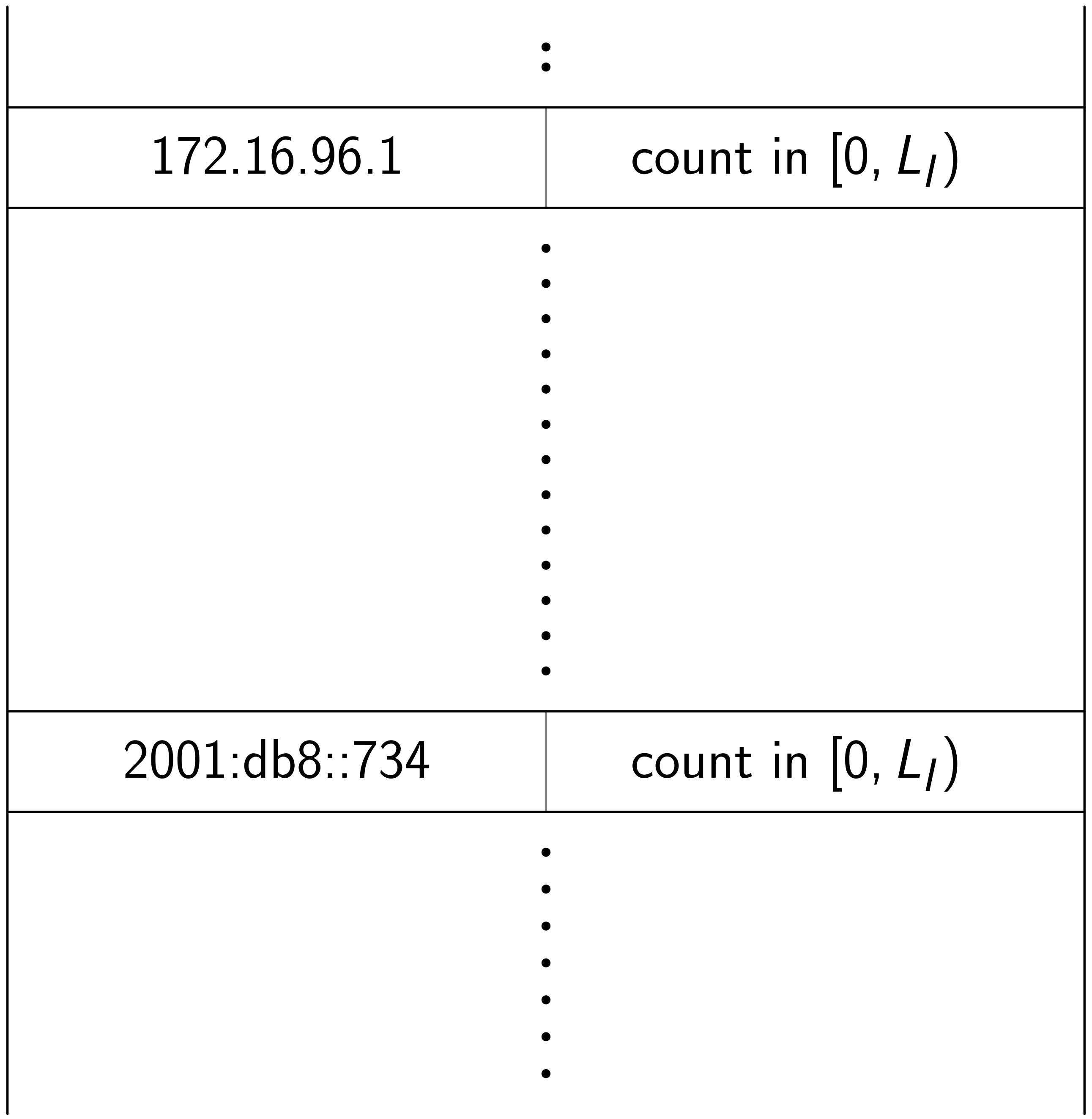
1. amplification att.
   - forged UDP source
   - answers >> queries
   - size limit (1232 B)
   - rate limiting
     - truncation
     - dropping
2. CPU overload
   - prioritization
     - all except plain UDP



- Mapping of addresses to counters – simplified.
- Values are counts of unrestricted queries, up to *instant* limit – max number of queries in a short period of time.

Limiting individual clients

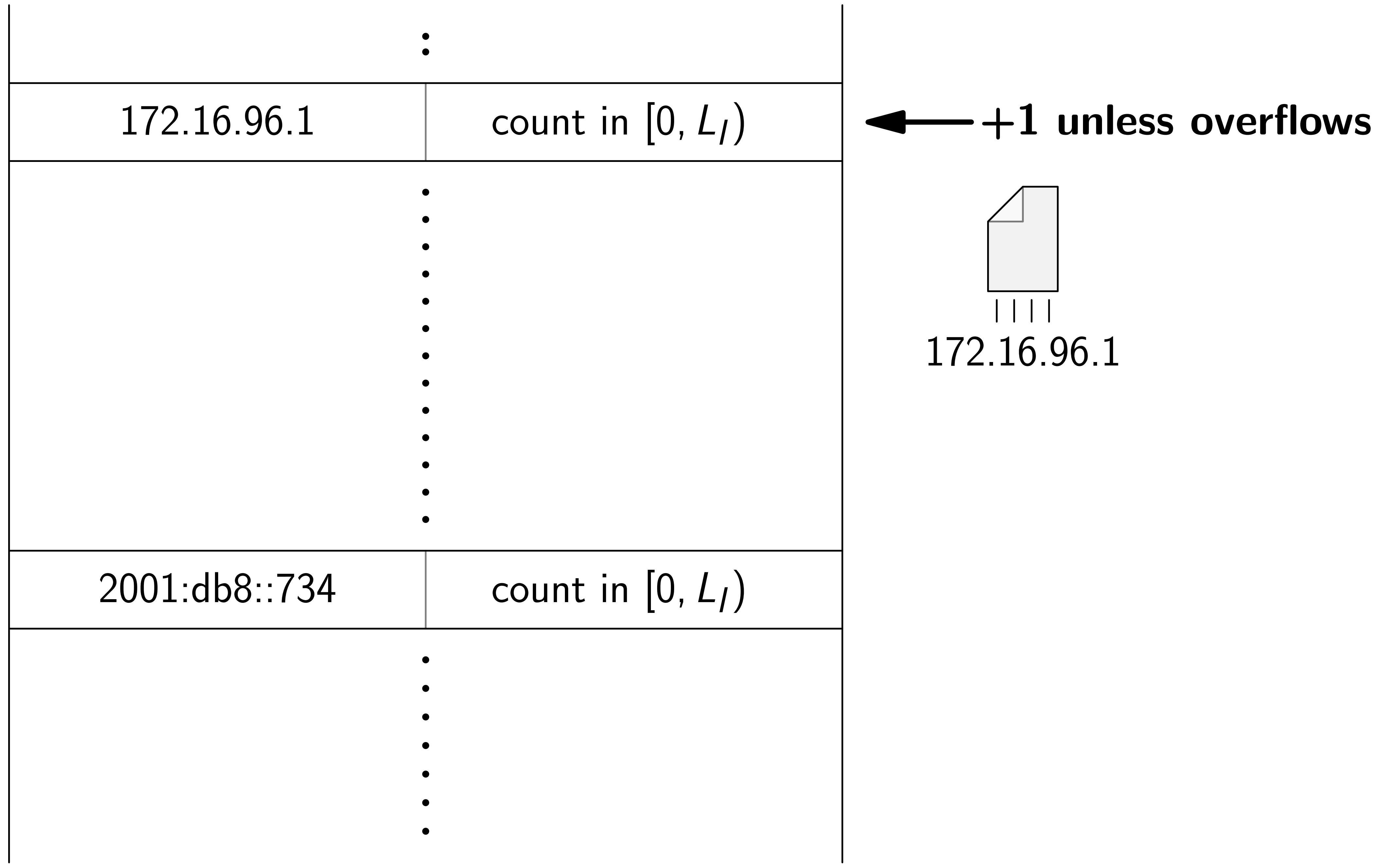
- **counters for addresses**
  - **instant limit L_I**



- Counting only unrestricted queries.

Limiting individual clients

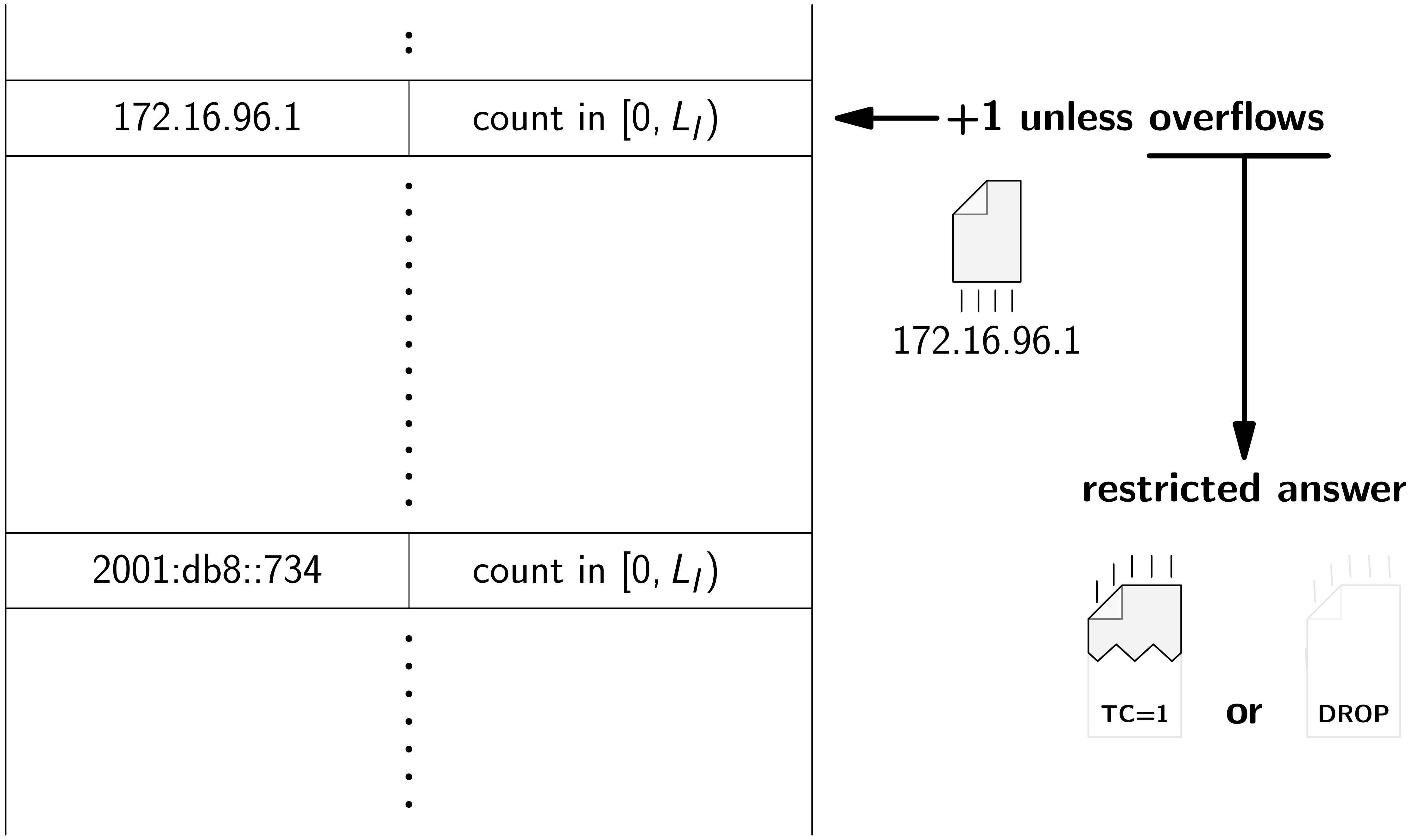
- **counters for addresses**
  - **instant limit L_I**



- Reaching limit -> restricted response.

Limiting individual clients

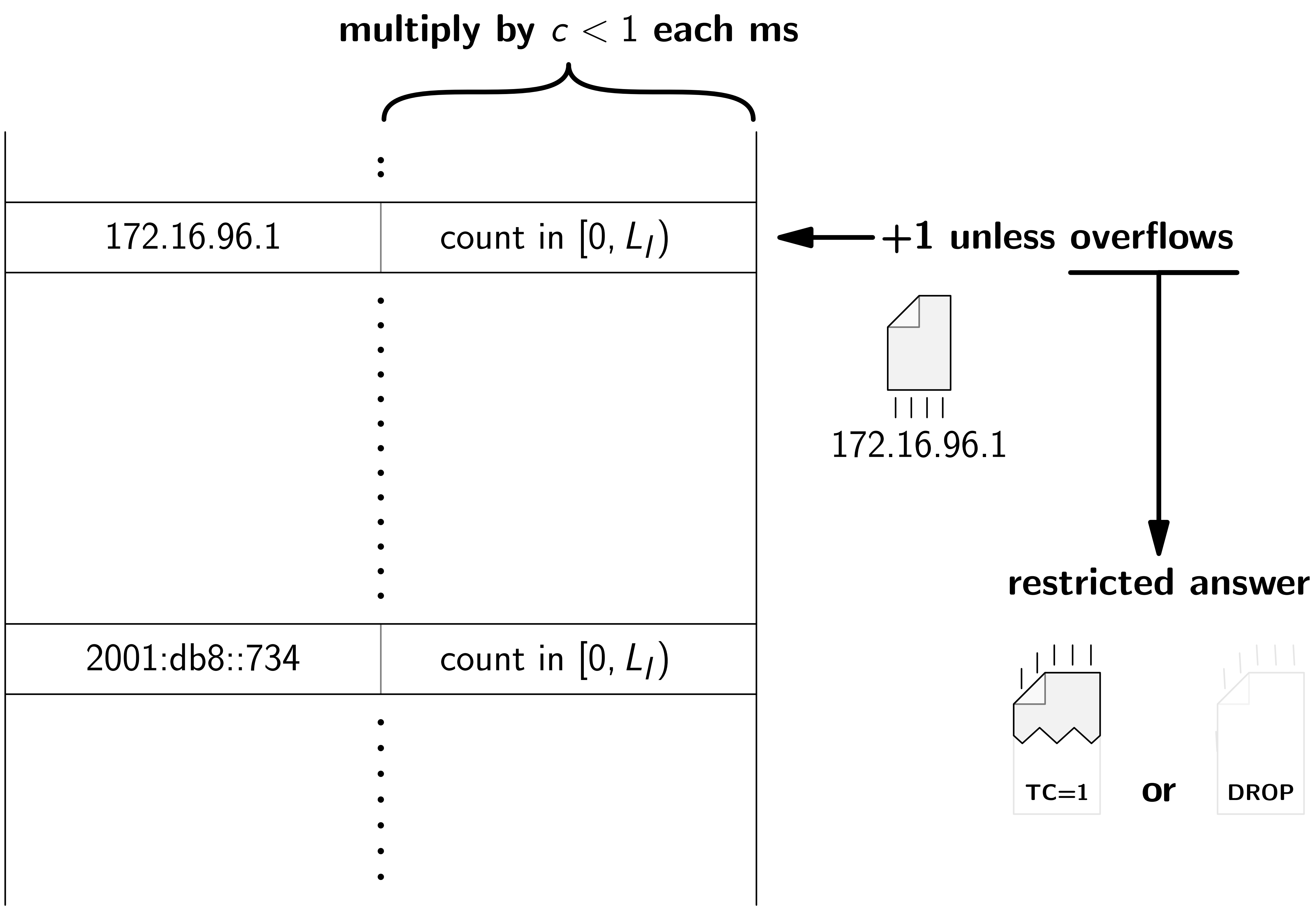
- **counters for addresses**
  - **instant limit L_I**



Limiting individual clients

- Decreasing by a constant fraction of its value each ms.
- Exponential decay, resembles radioactive decay.
- The speed of decreasing given by *rate* limit – allowed number of queries per unit of time in the long-term.

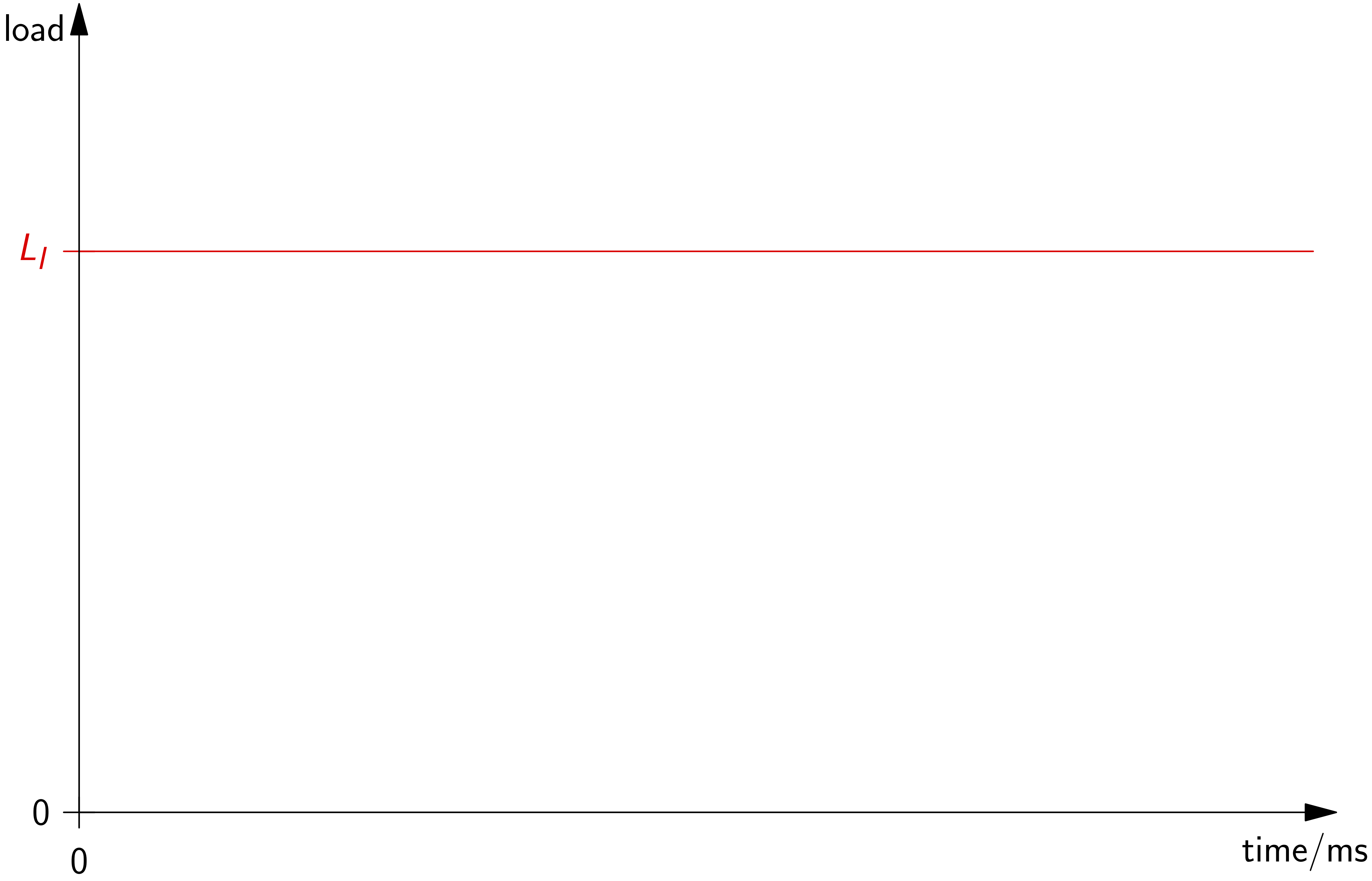
- counters for addresses
  - instant limit L_I
- **exponential decay**
  - **rate limit**



- The limiting is configured by two values: Instant limit and (long-term) Rate limit.
- instant limit – max value of the counter

Exponential decay

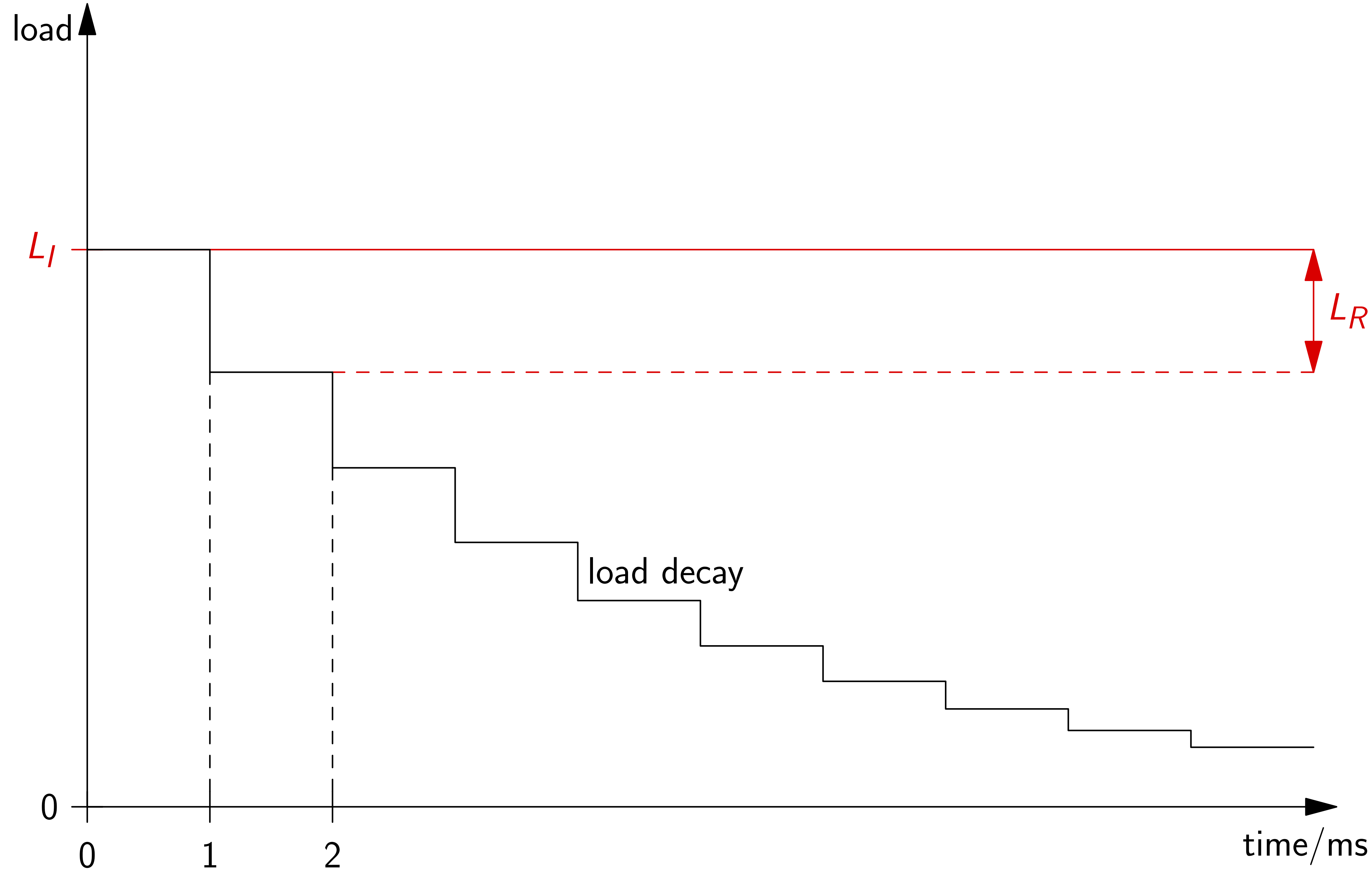
- **instant limit L_I**



Exponential decay

- decay after filling the counter
  - decreasing by constant fract. of the value each ms
- rate in ms – size of the first step
  - other steps lower

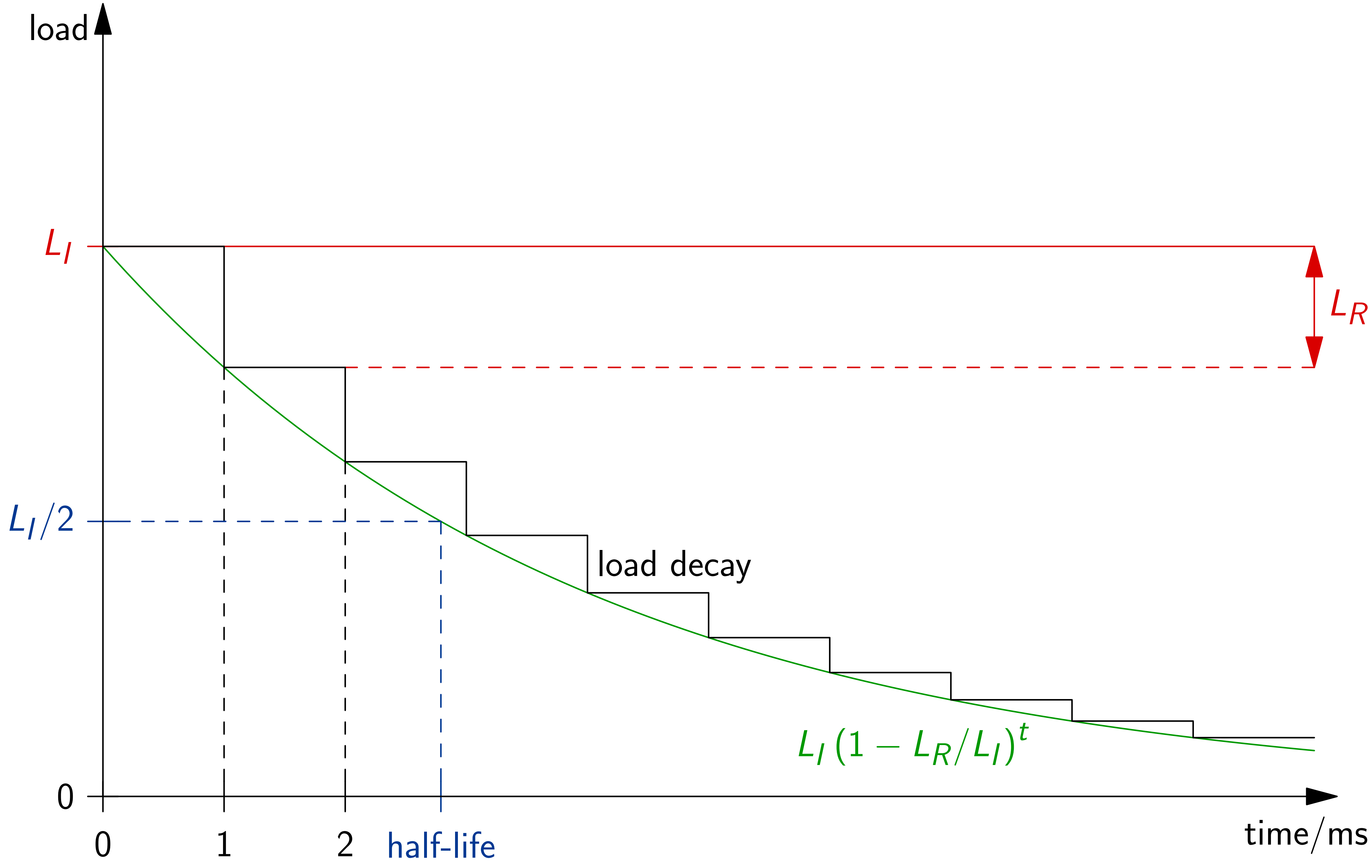
- instant limit L_I
- **rate limit L_R**
  - **per ms**



- (1/3)
- ratio of rate to instant gives half-life

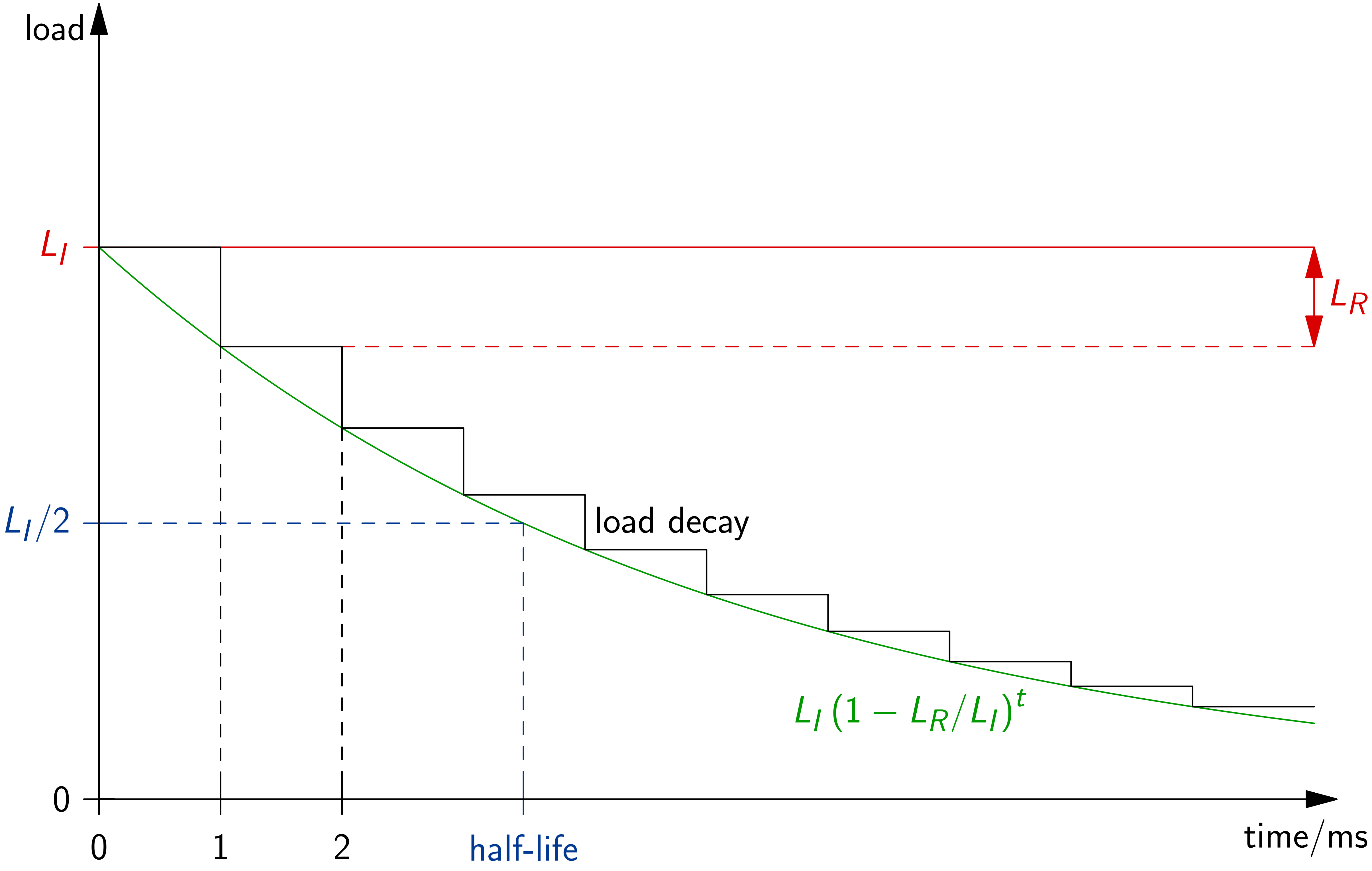
Exponential decay

- instant limit L_I
- rate limit L_R
  - per ms
- **half-life**



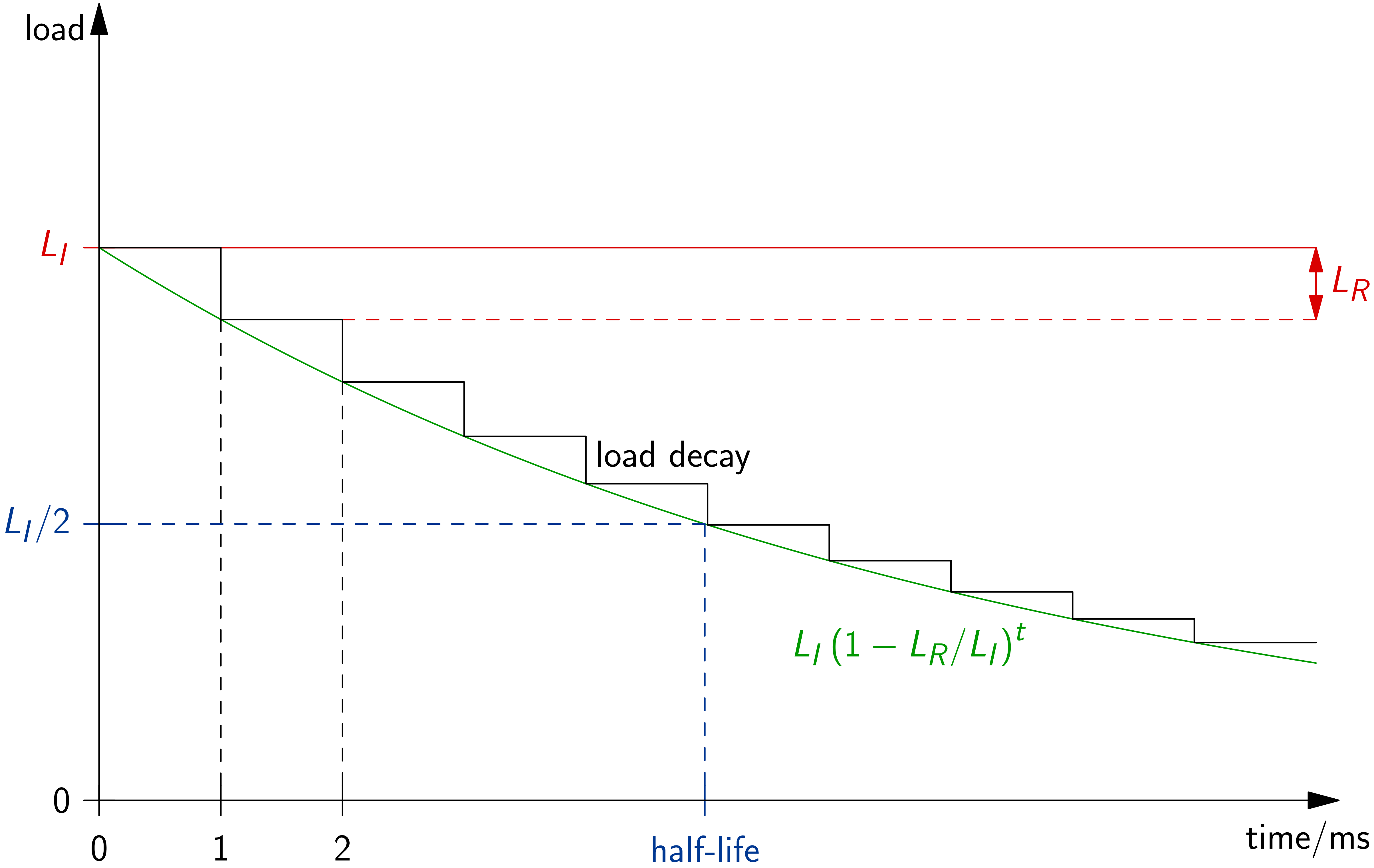
Exponential decay

- instant limit L_I
- rate limit L_R
  - per ms
- half-life



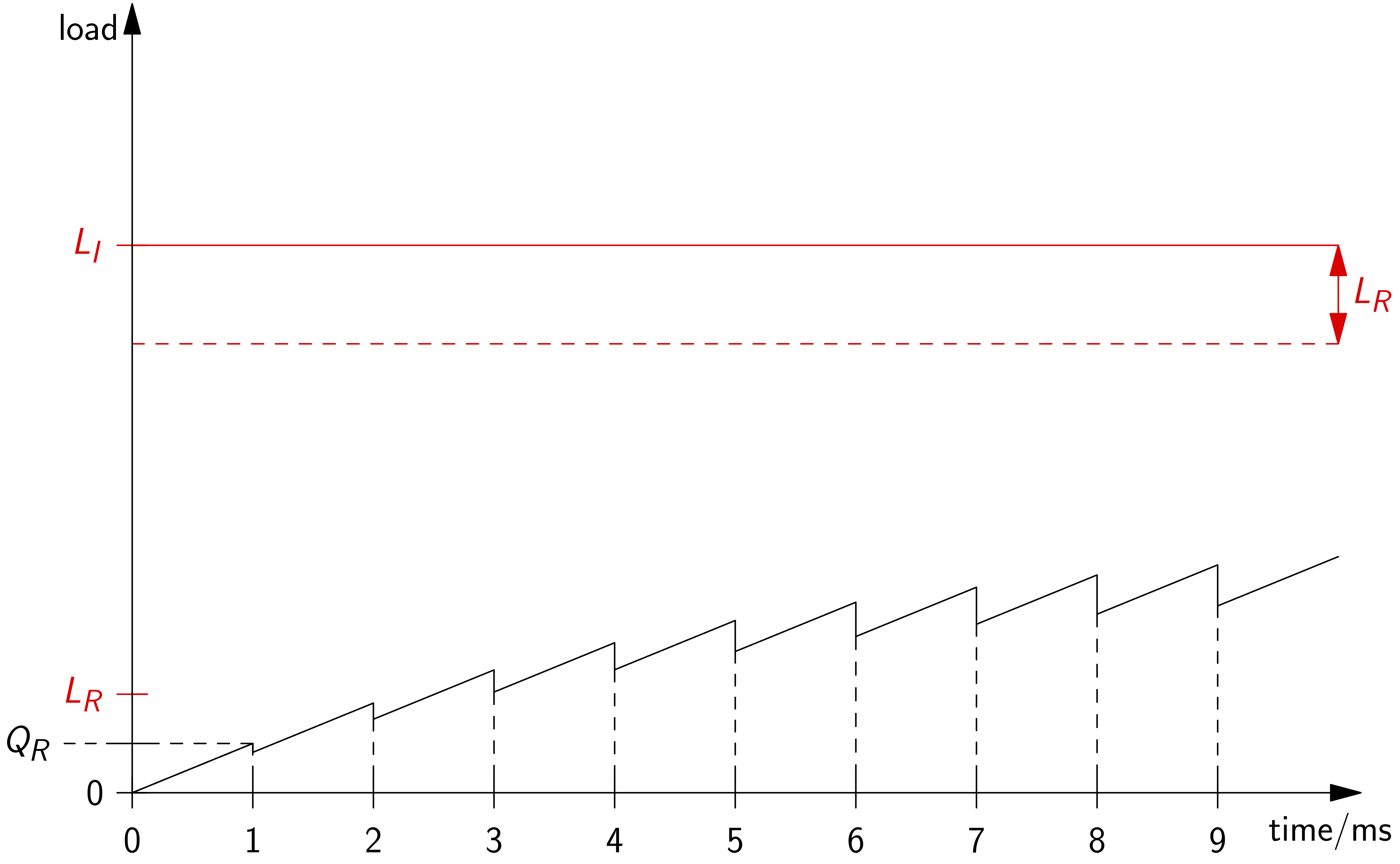
Exponential decay

- instant limit L_I
- rate limit L_R
  - per ms
- half-life



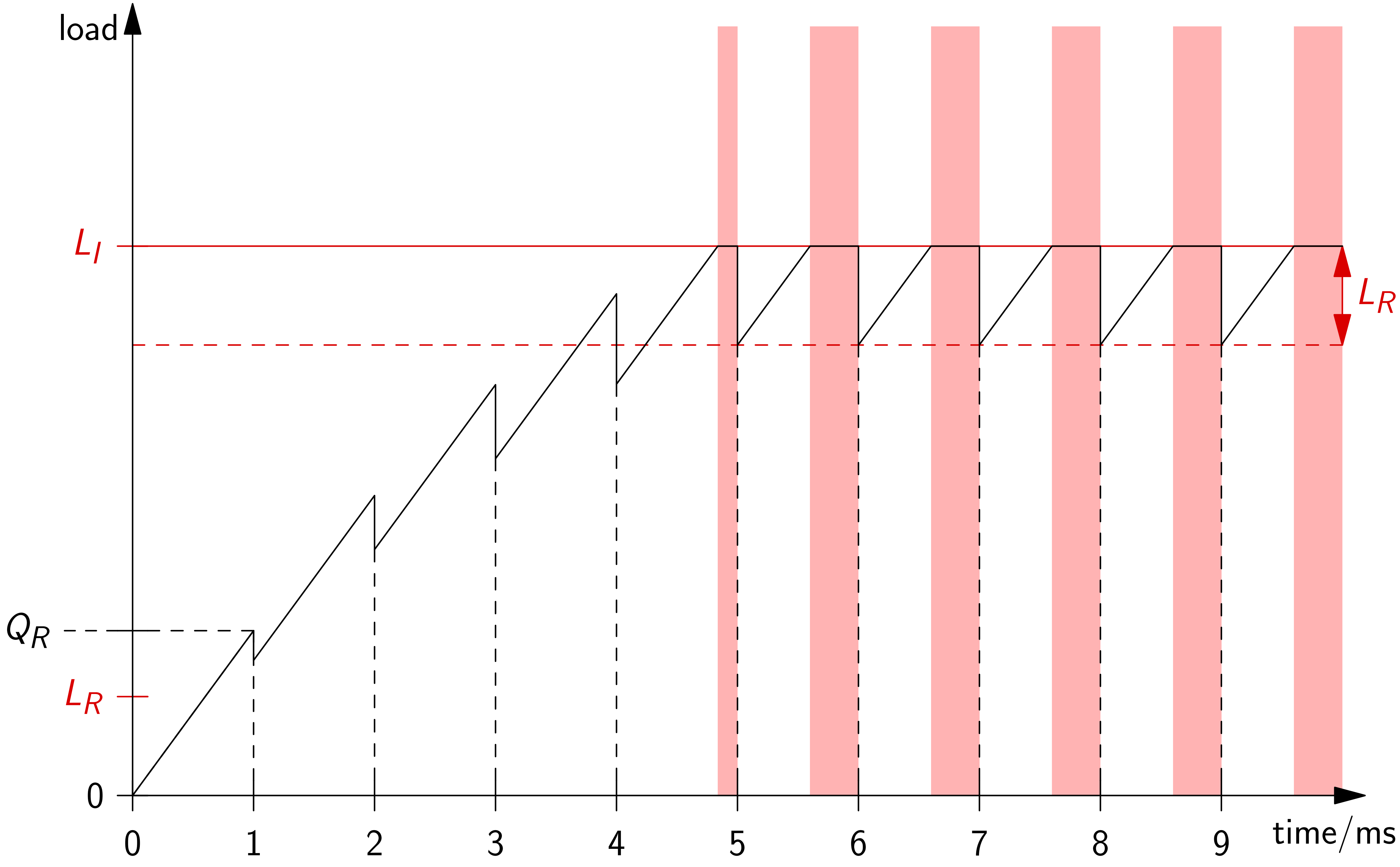
Constant query rate example

- instant limit L_I
- rate limit L_R
  - per ms
- half-life
- **query rate Q_R**
  - **per ms**



Constant query rate example

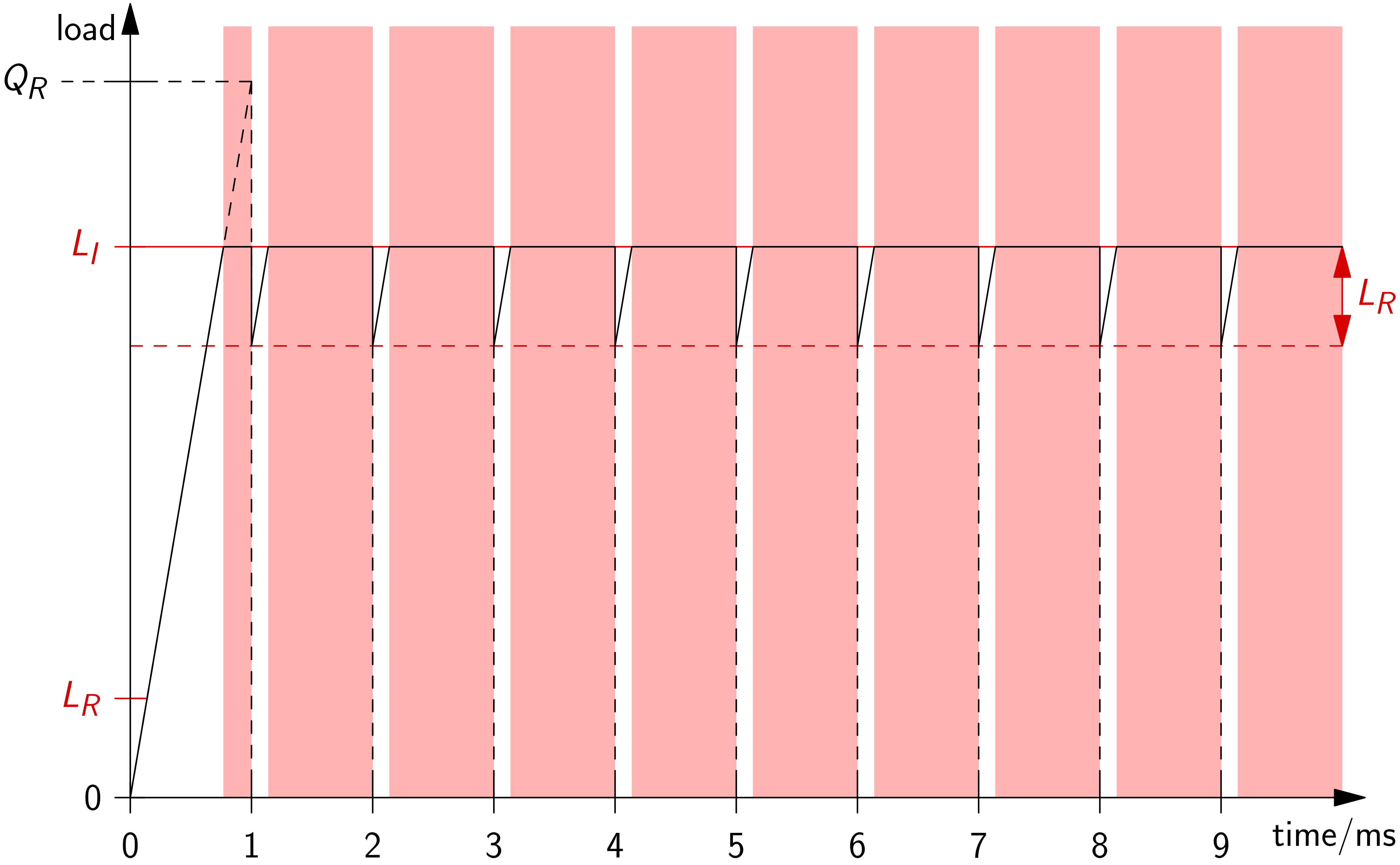
- instant limit L_I
- rate limit L_R
  - per ms
- half-life
- query rate Q_R
  - per ms



Constant query rate example

- (3/3)
- query rate above instant limit
  - restricted just after reaching L_I
  - then according to rate limit
- unrestricted queries -> something passes through, response rate is limited

- instant limit L_I
- rate limit L_R
  - per ms
- half-life
- query rate Q_R
  - per ms



Limiting networks

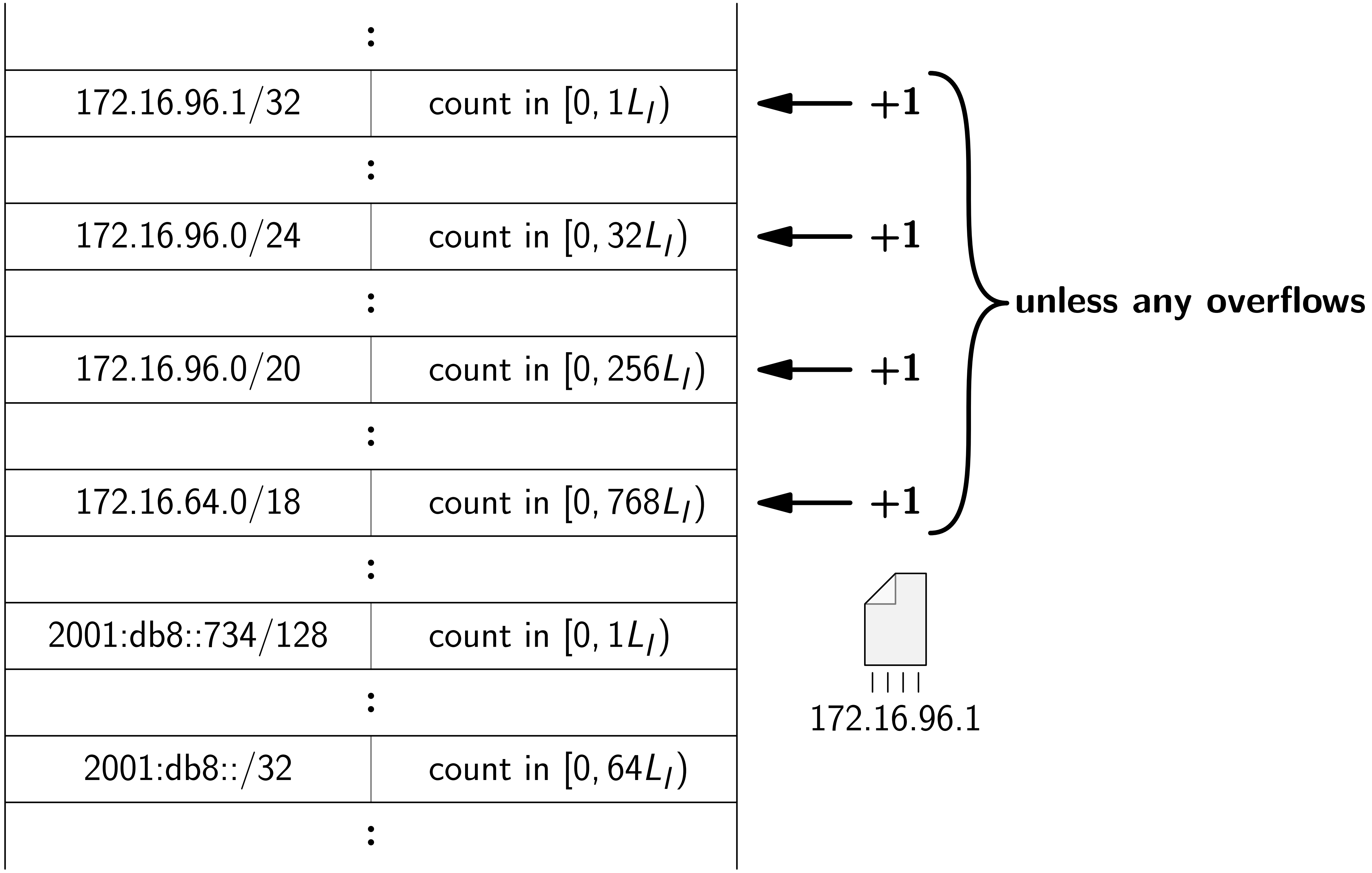
- Attackers won't play nice.
- Limiting individual IPs doesn't suffice.
- Usual approach: limit a single prefix size.
- We: maintain counters for *several* chosen prefixes.
- Constants multiplying limits based on prefix size.
- Shorter prefix -> larger network -> higher limits.
- Same multiplier for rate and instant -> half-life unchanged.

- IPv4
  - /32: 1
  - /24: 32
  - /20: 256
  - /18: 768
- IPv6
  - /128: 1
  - /64: 2
  - /56: 3
  - /48: 4
  - /32: 64

⋮	
172.16.96.1/**32**	count in $[0, \mathbf{1}L_l)$
⋮	
172.16.96.0/**24**	count in $[0, \mathbf{32}L_l)$
⋮	
172.16.96.0/**20**	count in $[0, \mathbf{256}L_l)$
⋮	
172.16.64.0/**18**	count in $[0, \mathbf{768}L_l)$
⋮	
2001:db8::734/**128**	count in $[0, \mathbf{1}L_l)$
⋮	
2001:db8::/**32**	count in $[0, \mathbf{64}L_l)$
⋮	

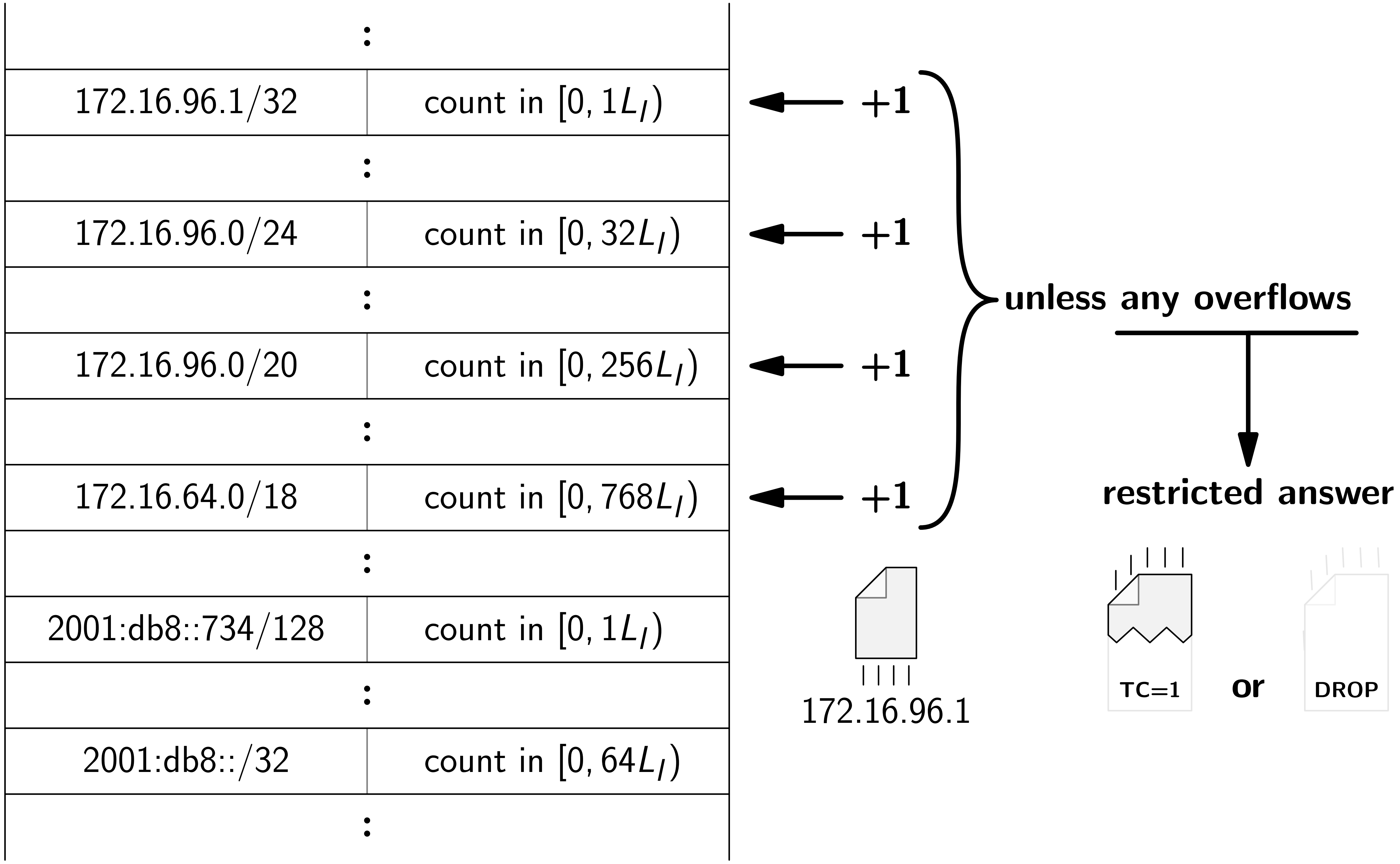
Limiting networks

- IPv4
  - /32: 1
  - /24: 32
  - /20: 256
  - /18: 768
- IPv6
  - /128: 1
  - /64: 2
  - /56: 3
  - /48: 4
  - /32: 64



Limiting networks

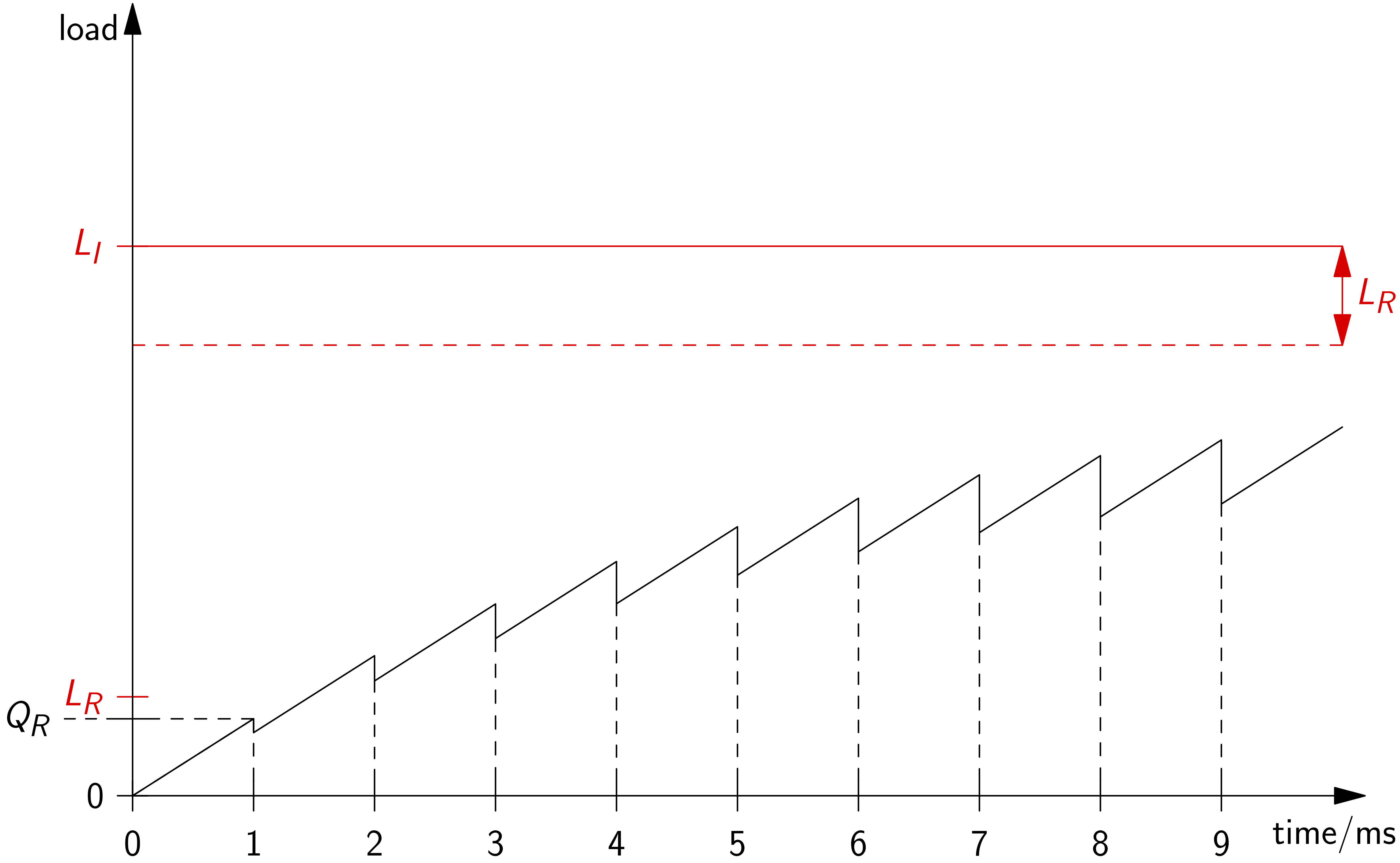
- IPv4
  - /32: 1
  - /24: 32
  - /20: 256
  - /18: 768
- IPv6
  - /128: 1
  - /64: 2
  - /56: 3
  - /48: 4
  - /32: 64



- So far only hard limits for dropping.
- Add lower instant and rate limits for truncating.

Multiple limits

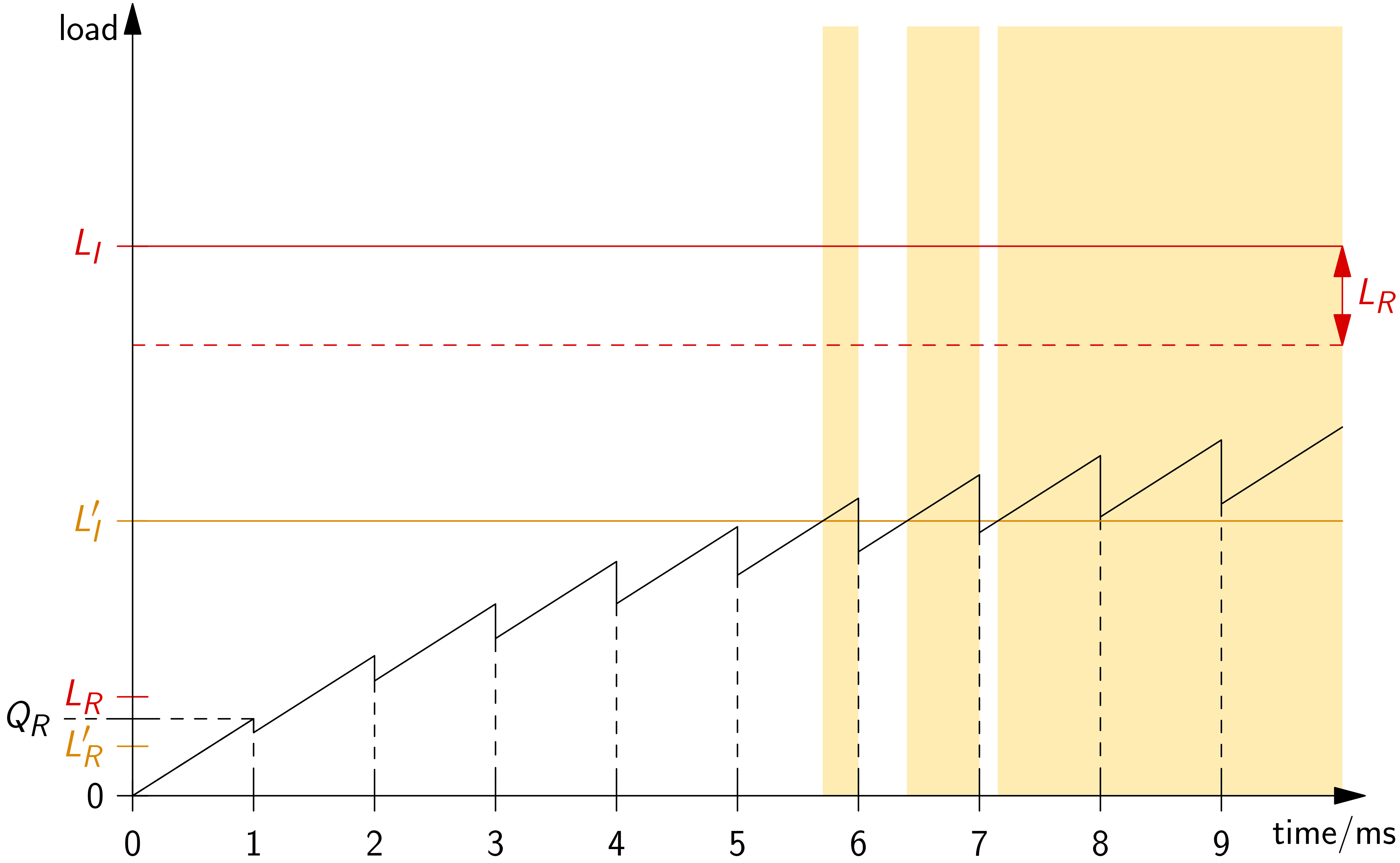
- hard limits
  - instant L_I
  - rate L_R per ms



Multiple limits

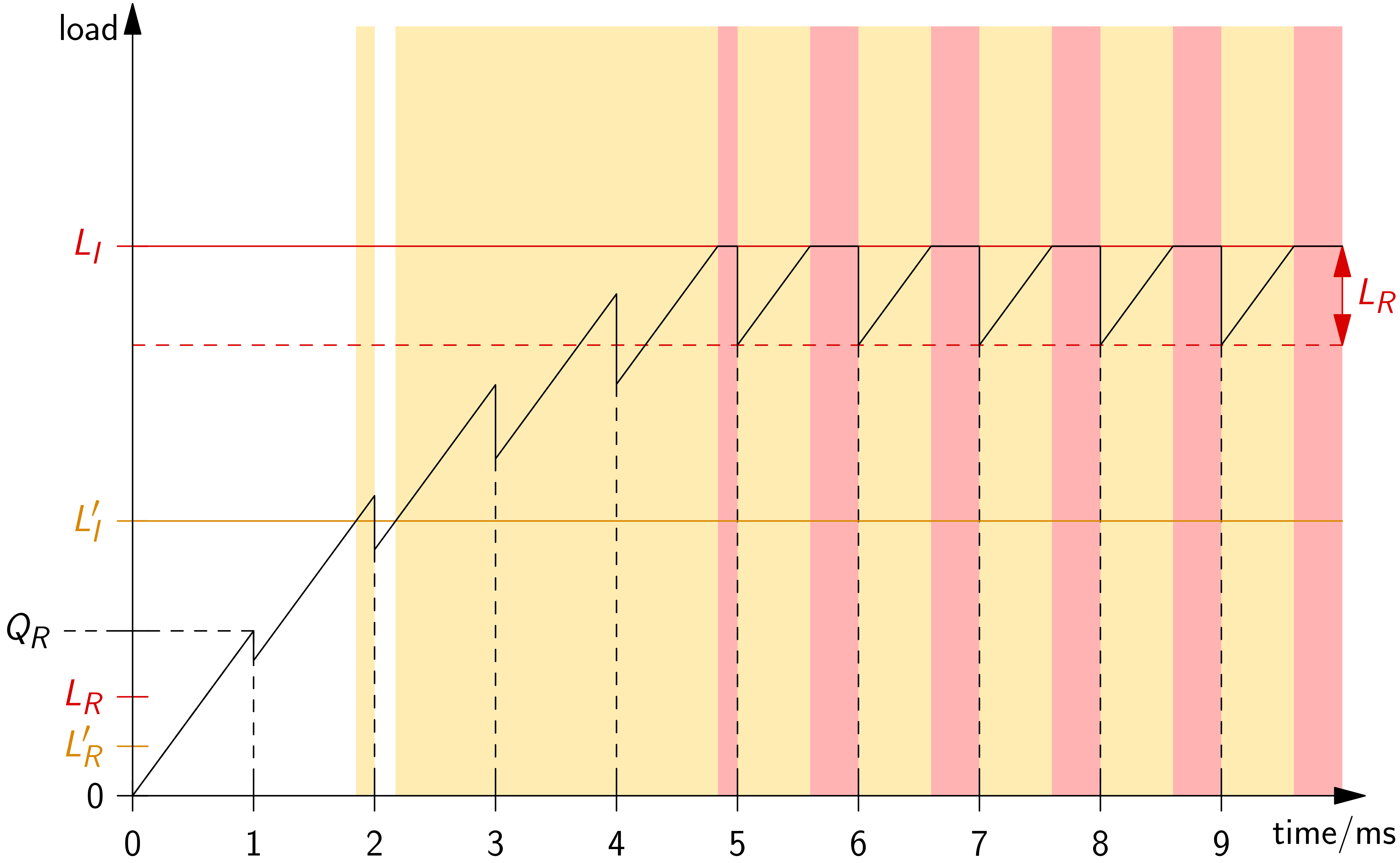
- (1/2)
- Incrementing also over soft limit – otherwise cannot reach hard limit.
- Everything truncated until user lowers query rate.
- On the chart query rate between soft and hard rate limit.

- hard limits
  - instant L_I
  - rate L_R per ms
- soft limits
  - instant L'_I
  - rate L'_R per ms



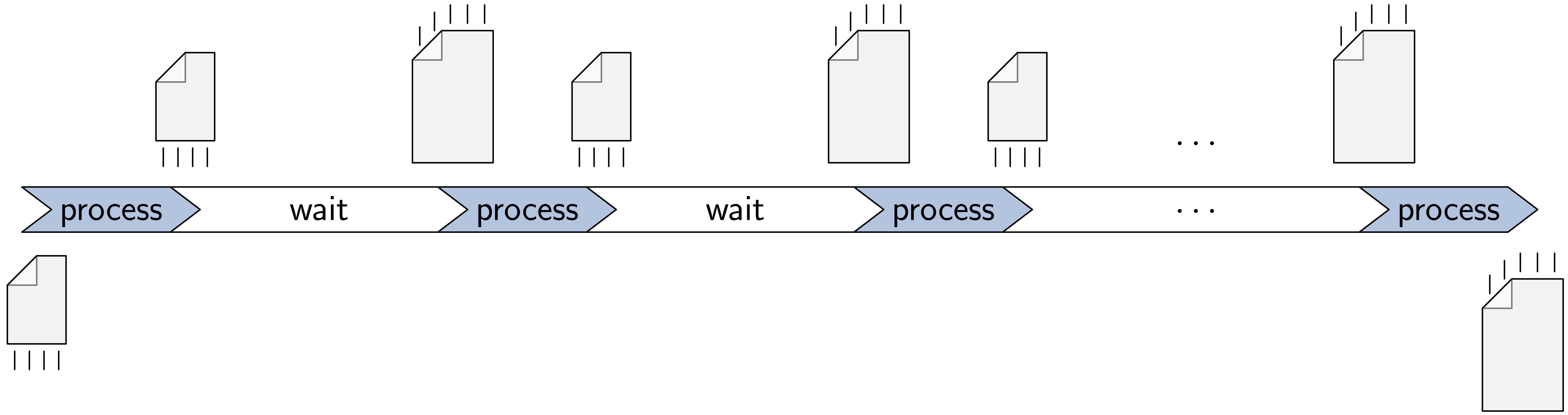
Multiple limits

- hard limits
  - instant L_I
  - rate L_R per ms
- soft limits
  - instant L'_I
  - rate L'_R per ms



Prioritization

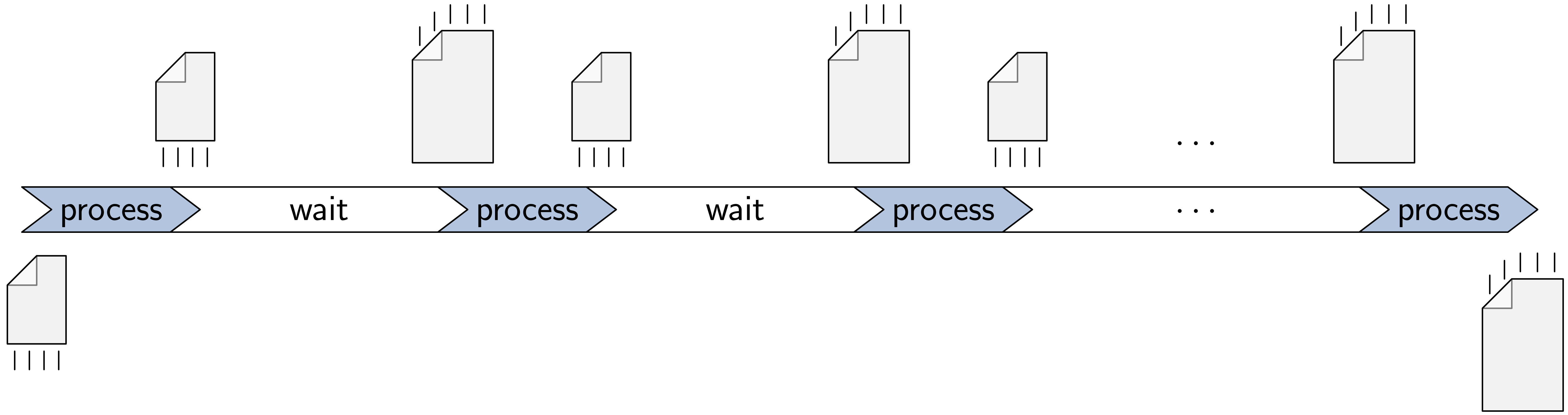
- not on plain UDP
- no configuration



Prioritization

- Even in legitimate traffic some queries are way more expensive than average.
- Aim: catch as much as possible while prioritizing users that don't overload our CPU.
- CPU time is measured, waiting not.

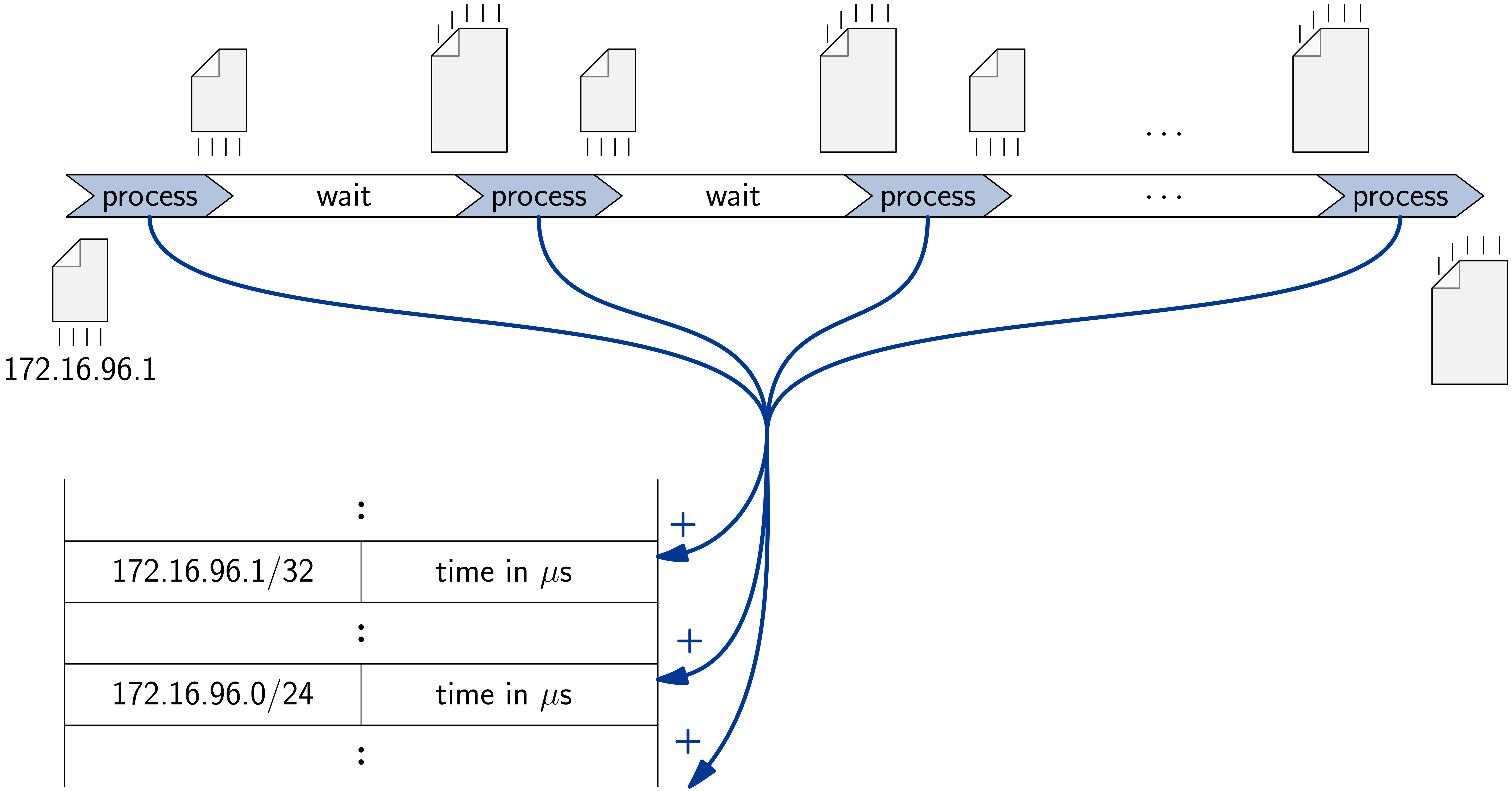
- not on plain UDP
- no configuration
- **measuring time**
  - **only cpu, no wait**



- Incrementing counters by time in μs.
  - different table instance
  - both addresses and networks

Prioritization

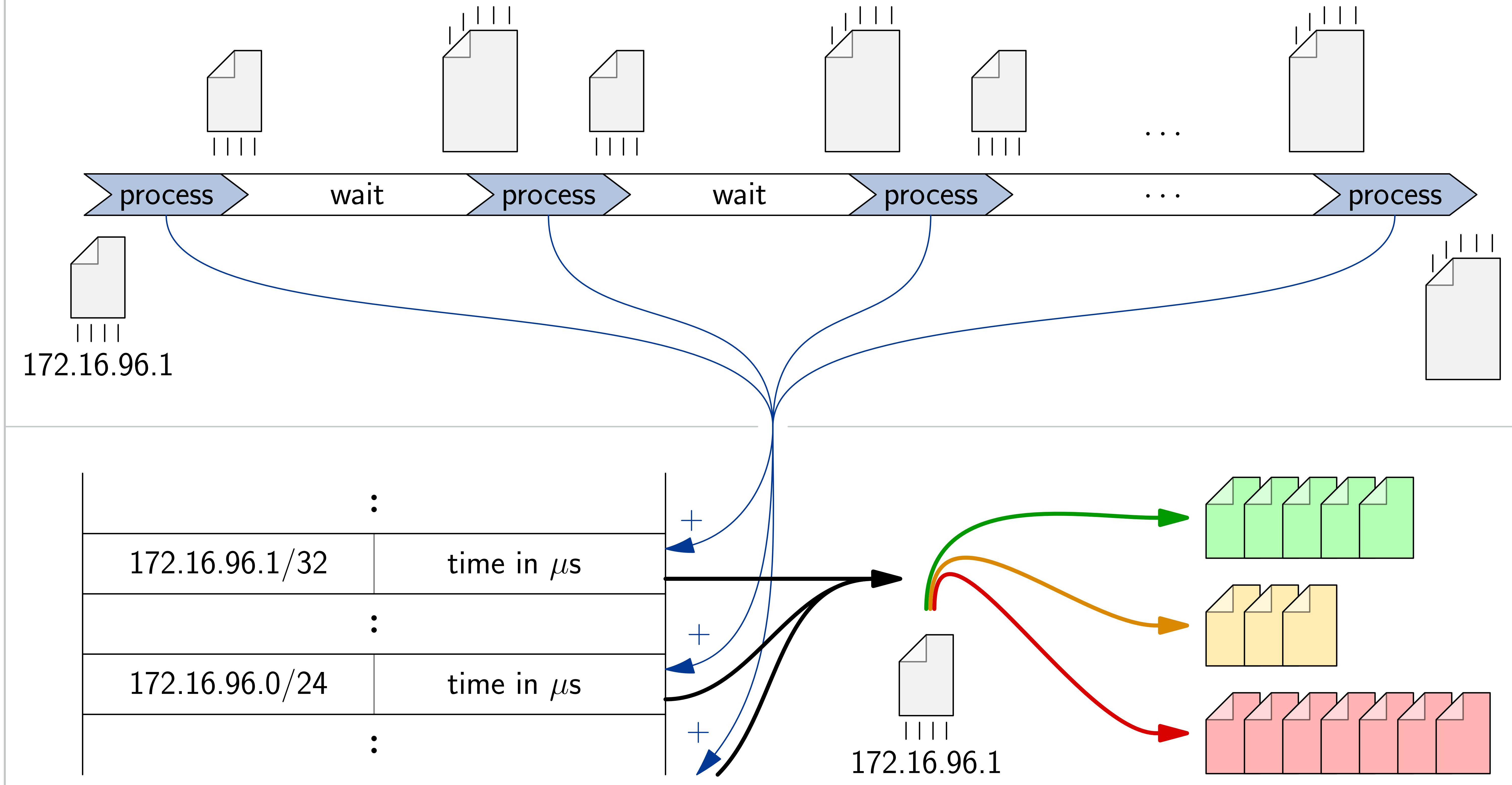
- not on plain UDP
- no configuration
- measuring time
  - only cpu, no wait
  - **add to table values**



- Multiple soft limits for different priorities.
  - A queue for each priority.
  - May be deferred multiple times on priority decrease.

Prioritization

- not on plain UDP
- no configuration
- measuring time
  - only cpu, no wait
  - add to table values
- **defer to queues**
  - **based on values**

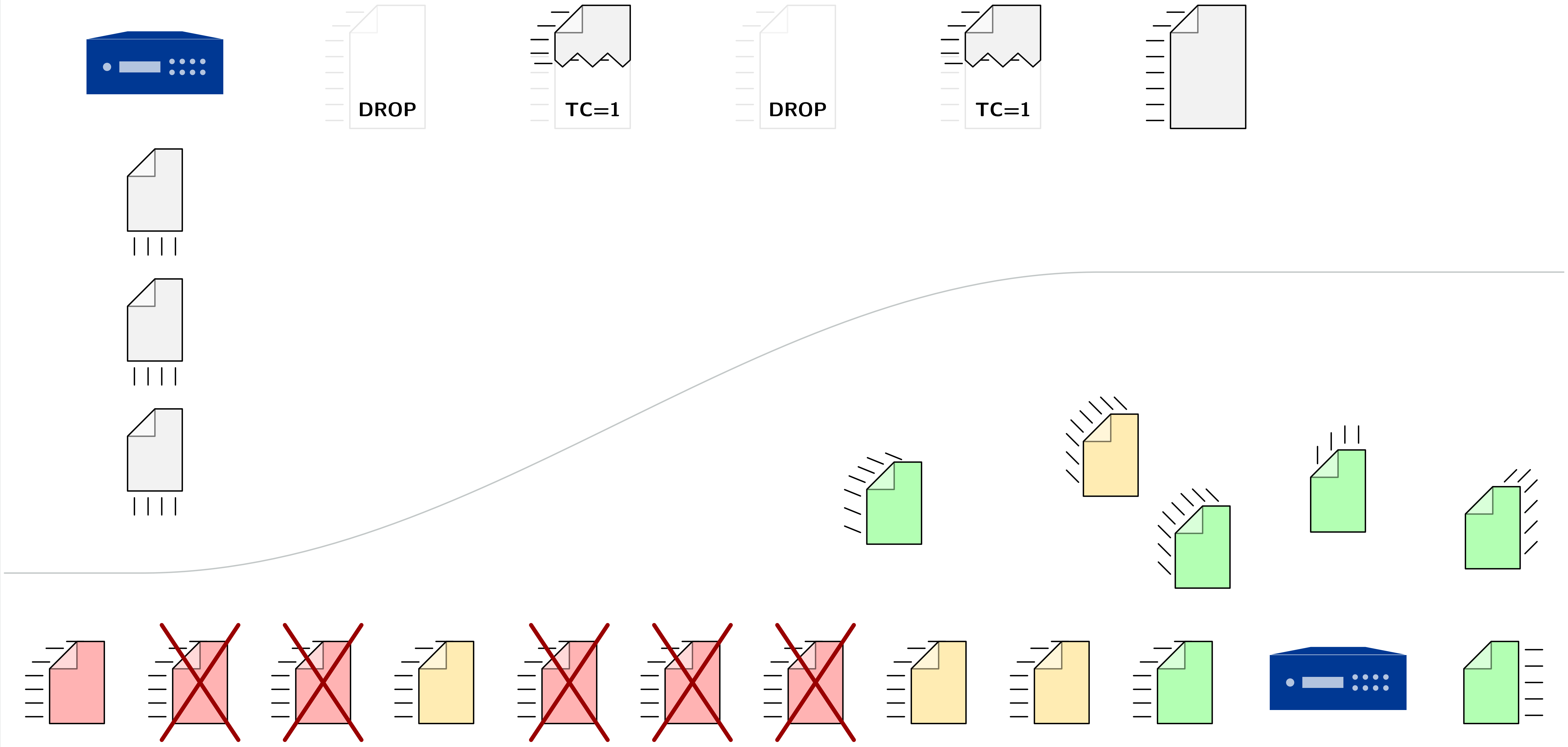


 cz.nic CZ DOMAIN REGISTRY

Finishing with a copy of the overview slide.

Final overview

1. **amplification att.**
   - forged UDP source
   - answers >> queries
   - size limit (1232 B)
   - rate limiting
     - truncation
     - dropping
2. **CPU overload**
   - prioritization
     - all except plain UDP



Appendix

now extra slides not planned for LinuxDays

Implementation

- **hashing**

$\text{hash}(172.16.96.1/32) = 1011110110000110101011000011010100111010011101101001000001011111$

⋮	
172.16.96.1/32	3 $\in [0, 1L_I)$
⋮	
172.16.96.0/24	15.34 $\in [0, 32L_I)$
⋮	
172.16.96.0/20	123 $\in [0, 256L_I)$
⋮	
2001:db8::734/128	7.569 $\in [0, 1L_I)$
⋮	
2001:db8::/32	33.21 $\in [0, 64L_I)$
⋮	

- Use buckets with several (15) most important records.
- Still low number of buckets will have many collisions.

Implementation

- hashing
  - **buckets**

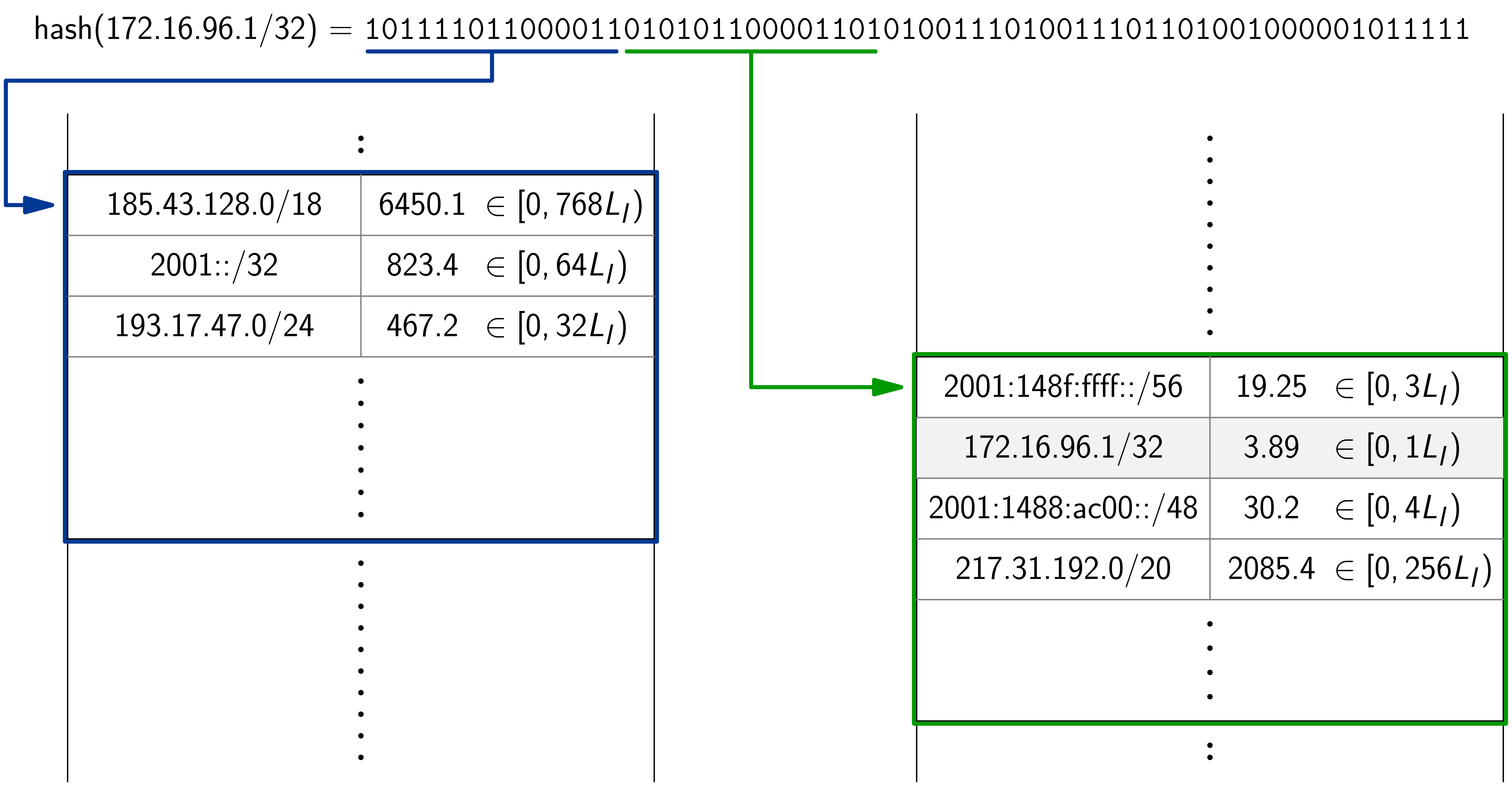
$\text{hash}(172.16.96.1/32) = \underline{1011110110000110}101011000011010100111010011101101001000001011111$

⋮	
185.43.128.0/18	$6450.1 \in [0, 768L_l)$
2001::/32	$823.4 \in [0, 64L_l)$
193.17.47.0/24	$467.2 \in [0, 32L_l)$
2001:148f:ffff::/56	$19.25 \in [0, 3L_l)$
172.16.96.1/32	$3.89 \in [0, 1L_l)$
⋮	

- Two tables, hashed independently.
- The probability of collision in both of them is much smaller.

Implementation

- hashing
  - buckets
  - **two tables**

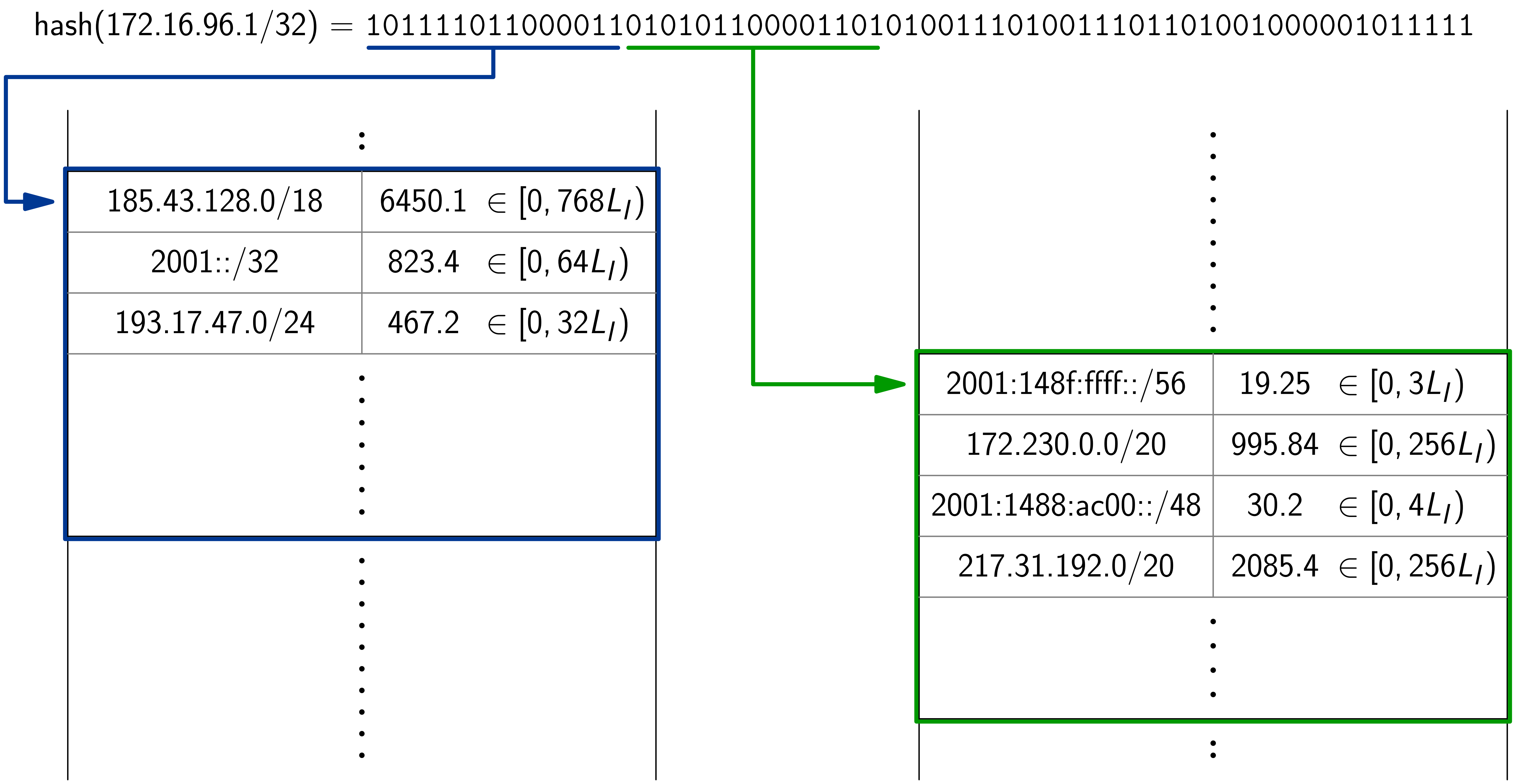


CZ DOMAIN REGISTRY

- Let’s say our address is not there.

Implementation

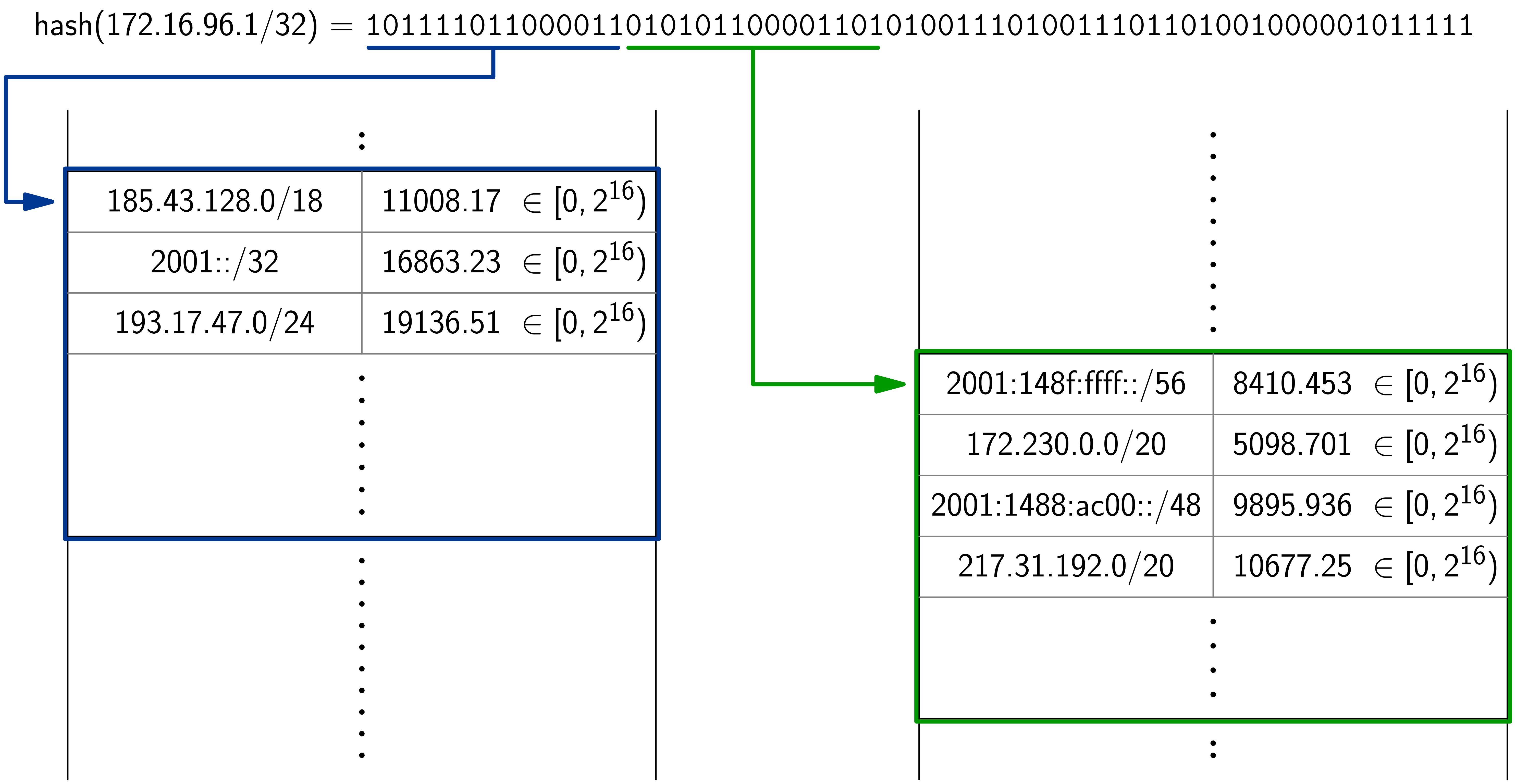
- hashing
  - buckets
  - two tables
- **evicting**



Implementation

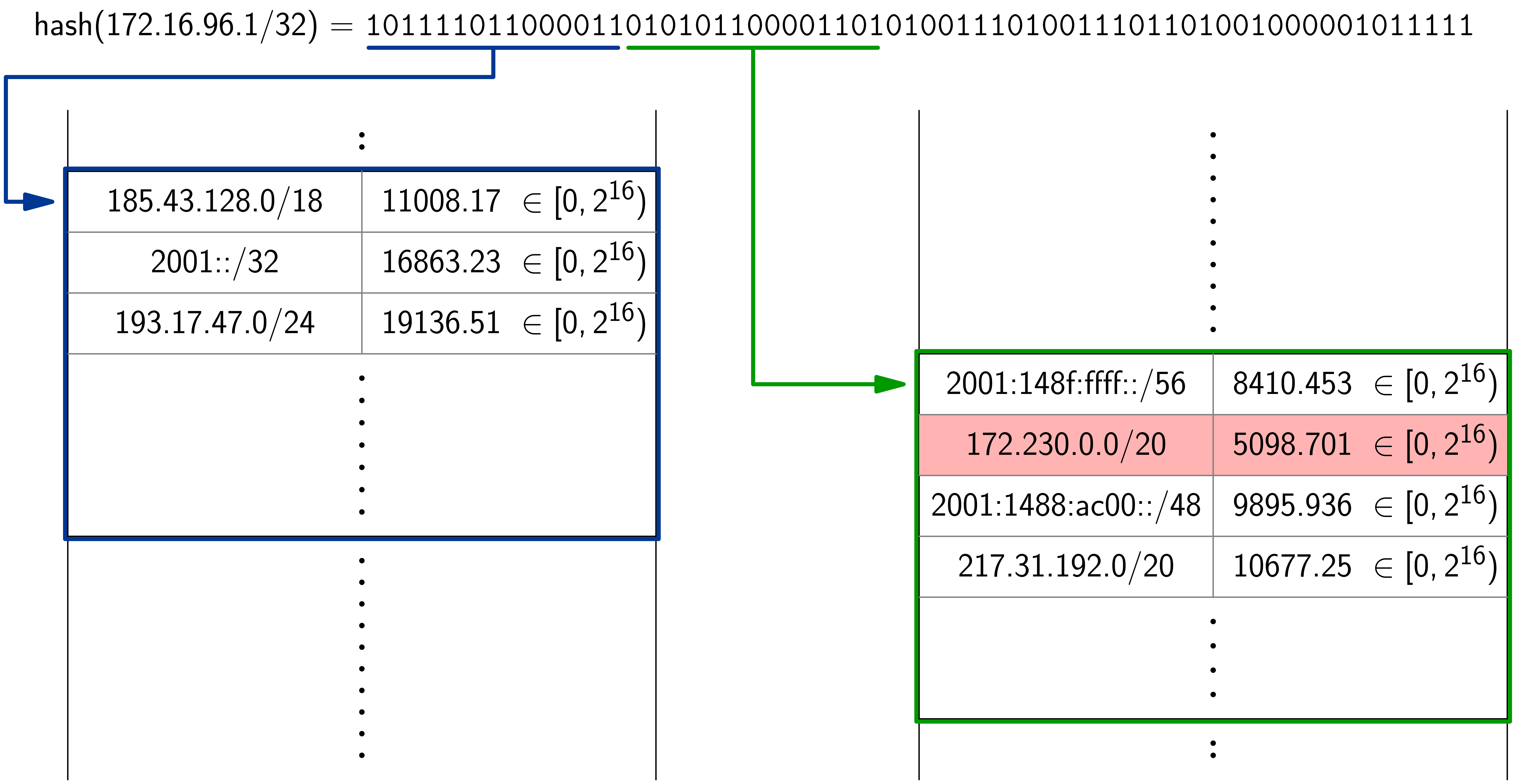
- Normalize to the same limit.
- It allows comparing values across different prefix length – gives us the notion of their importance.

- hashing
  - buckets
  - two tables
- evicting
  - **normalized limits**



Implementation

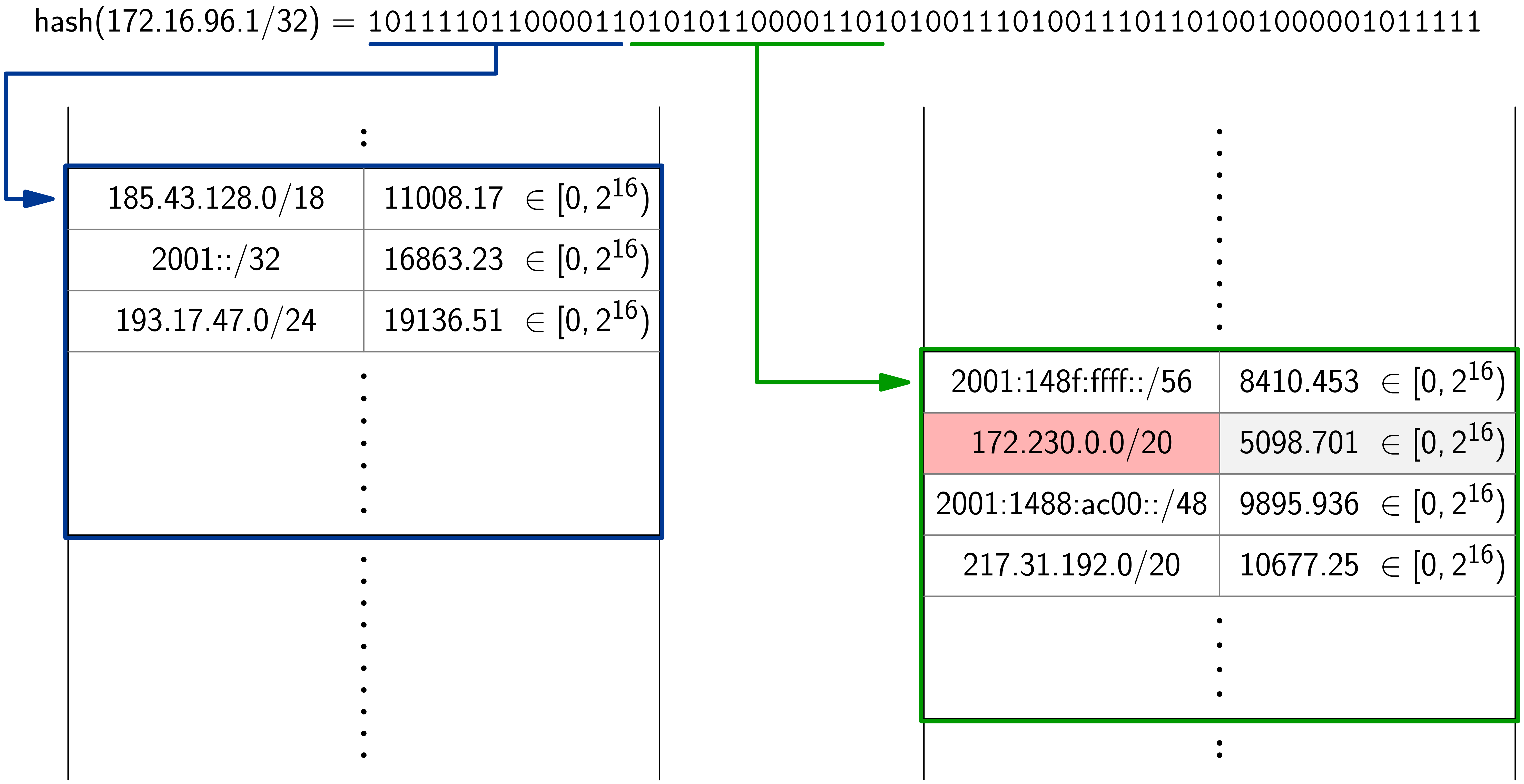
- hashing
  - buckets
  - two tables
- evicting
  - normalized limits
  - **choosing minimal**



Implementation

- In fact, evict only the label keeping value.
- Multiple items evicting each other share the value instead of zeroing.
- Leads to similar behavior as CountMin sketches, on overloading.

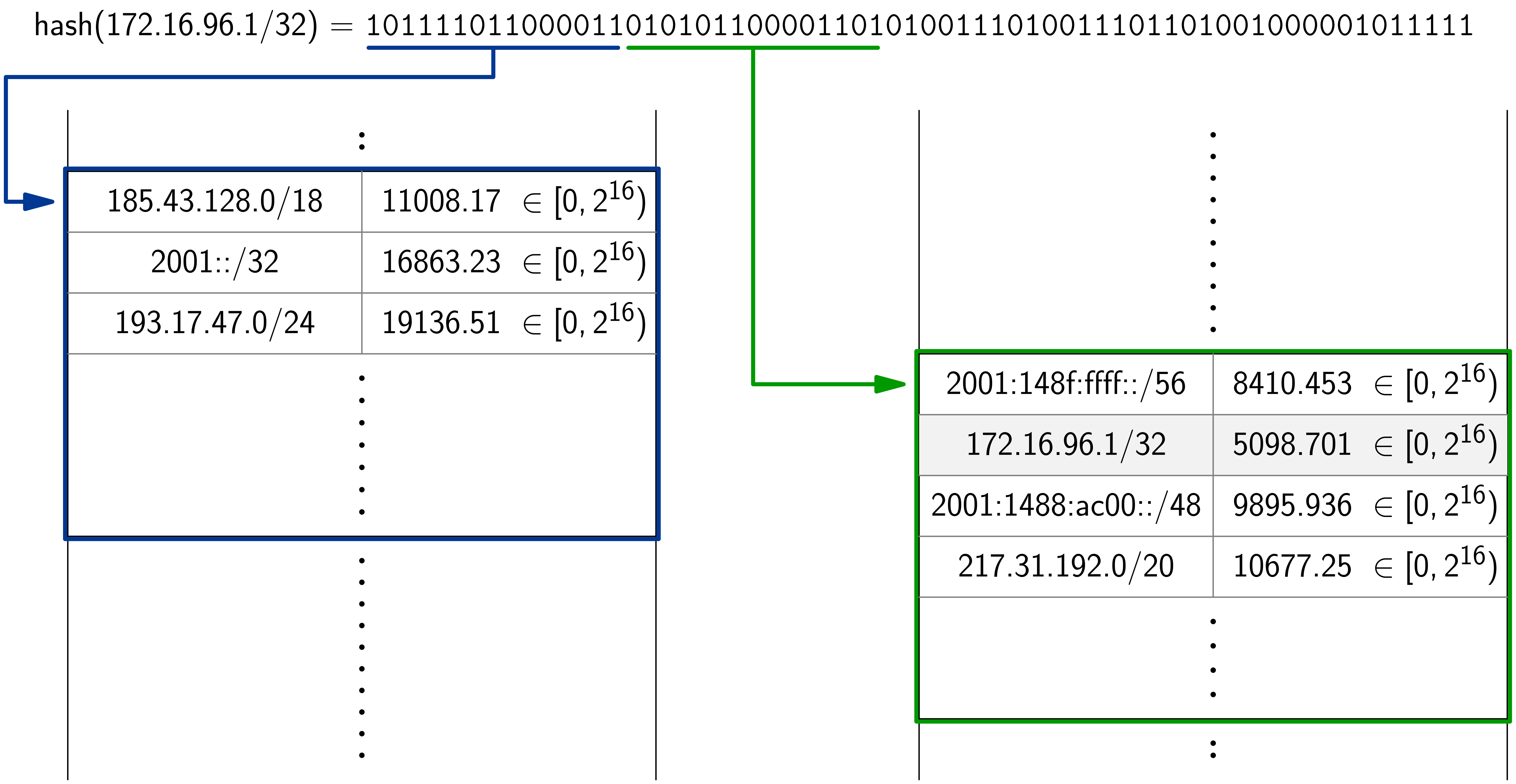
- hashing
  - buckets
  - two tables
- evicting
  - normalized limits
  - choosing minimal
  - **keeping value**



- Decay only on access, not each ms.

Implementation

- hashing
  - buckets
  - two tables
- evicting
  - normalized limits
  - choosing minimal
  - keeping value
- **lazy decay**



- Store timestamp of last decay in each bucket.
- Perform decay on all bucket items.

Implementation

- hashing
  - buckets
  - two tables
- evicting
  - normalized limits
  - choosing minimal
  - keeping value
- **lazy decay**

$$\text{hash}(172.16.96.1/32) = 1011110110000110101011000011010100111010011101101001000001011111$$

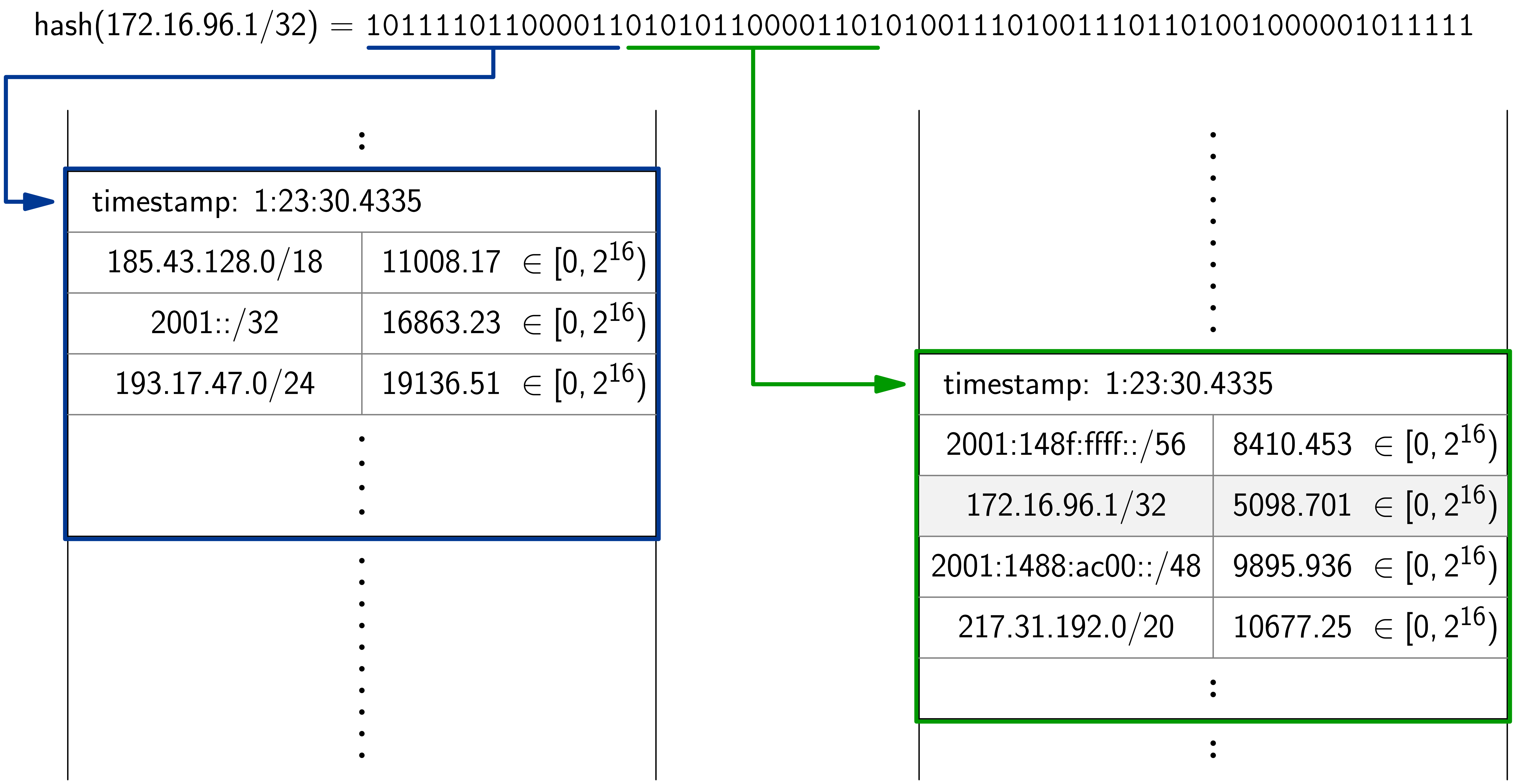
timestamp: 1:23:30.4335	
185.43.128.0/18	11008.17 $\in [0, 2^{16})$
2001::/32	16863.23 $\in [0, 2^{16})$
193.17.47.0/24	19136.51 $\in [0, 2^{16})$
⋮	

timestamp: 1:23:30.4335	
2001:148f:ffff::/56	8410.453 $\in [0, 2^{16})$
172.16.96.1/32	5098.701 $\in [0, 2^{16})$
2001:1488:ac00::/48	9895.936 $\in [0, 2^{16})$
217.31.192.0/20	10677.25 $\in [0, 2^{16})$
⋮	

- How to fit it in memory saving as much space as possible.

Implementation

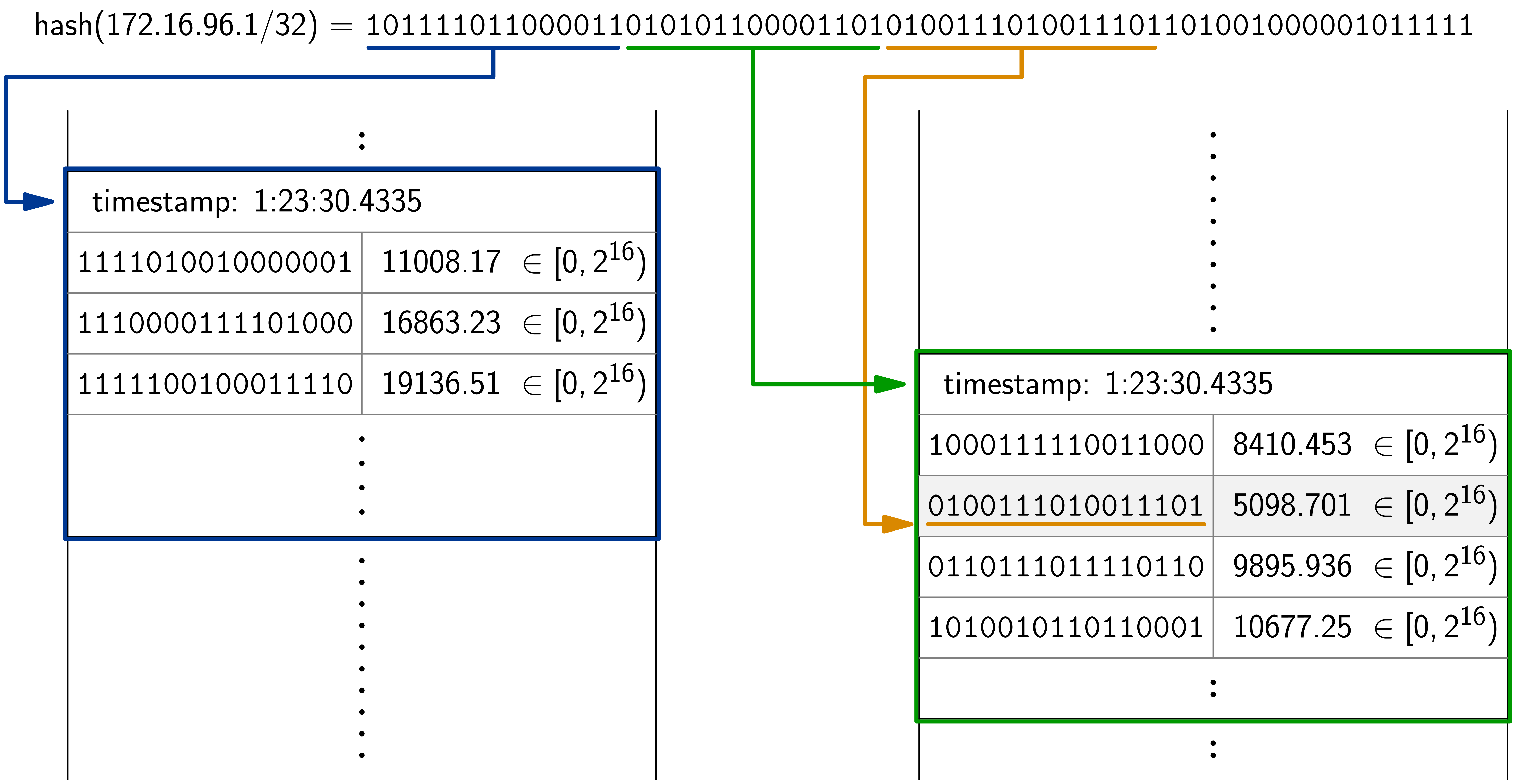
- hashing
  - buckets
  - two tables
- evicting
  - normalized limits
  - choosing minimal
  - keeping value
- lazy decay
- **memory layout**



Implementation

- Addresses too long, store just another part of their hash (16 bits).
- Collisions may cause sharing counters, but they are very infrequent.

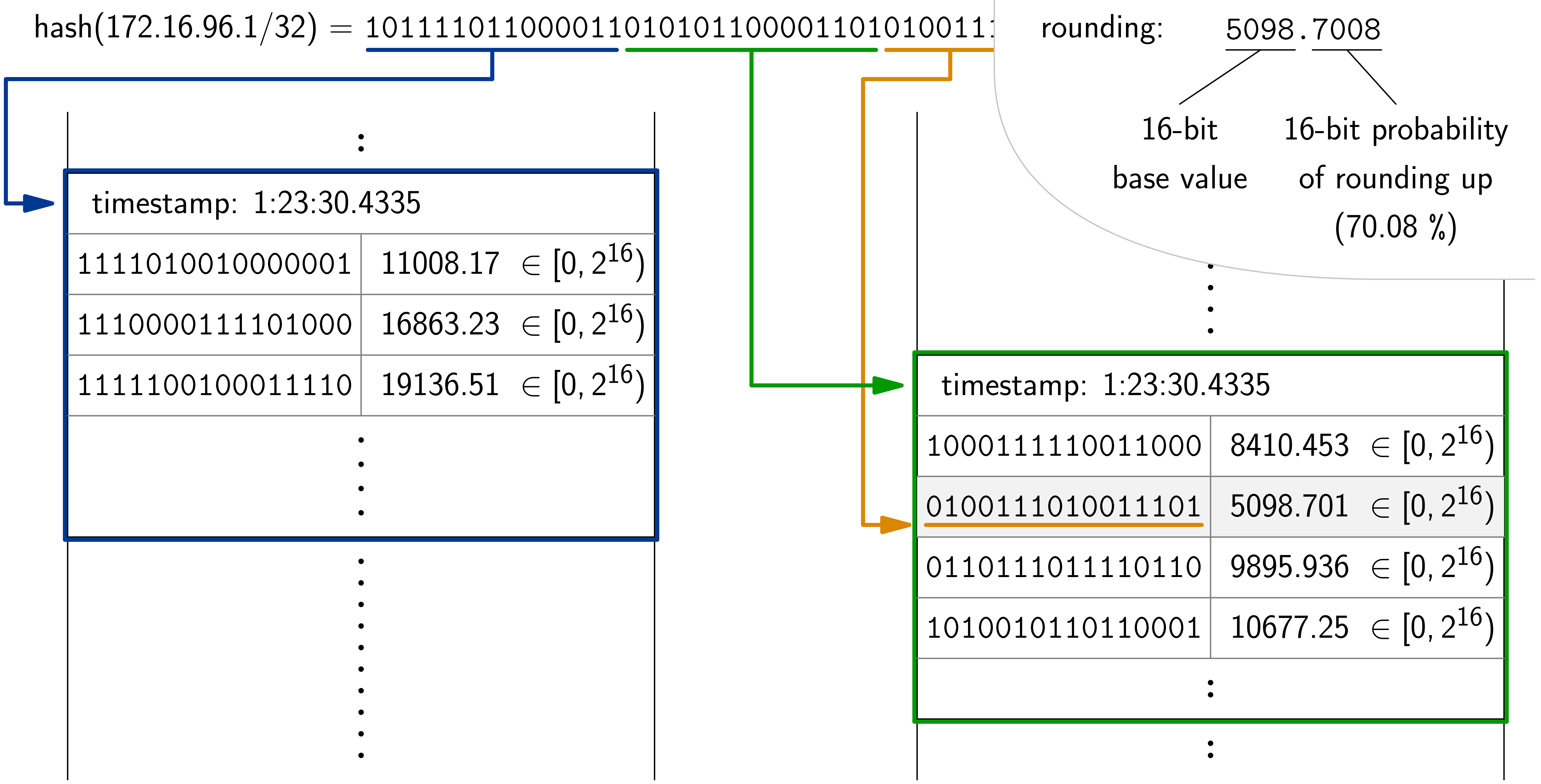
- hashing
  - buckets
  - two tables
- evicting
  - normalized limits
  - choosing minimal
  - keeping value
- lazy decay
- memory layout
  - **hashed labels**



Implementation

- Store just the whole part of values, > but rounded probabilistically by the fractional part.
- All calculations 32-bit – 16 in integer part, 16 in fractional.
- Before storing the integer part, use the fractional part as probability of rounding up.
  - Compare the 16 low significant bits to rnd.

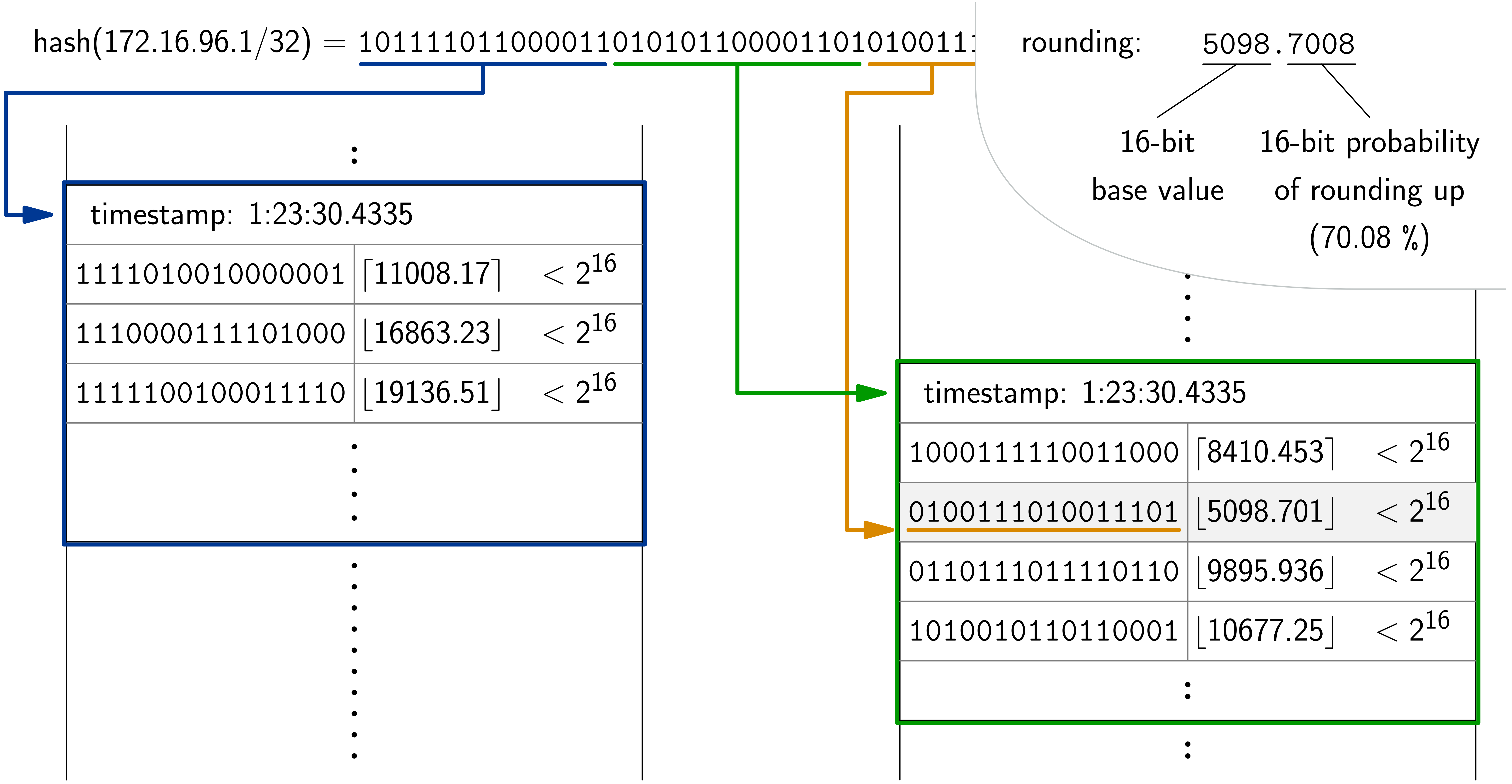
- hashing
  - buckets
  - two tables
- evicting
  - normalized limits
  - choosing minimal
  - keeping value
- lazy decay
- memory layout
  - hashed labels
  - **prob. rounding**



- So we have 16-bit values, but can increment even by much smaller fractions.
- Still very precise – 2^16 ones required to perform limiting.

Implementation

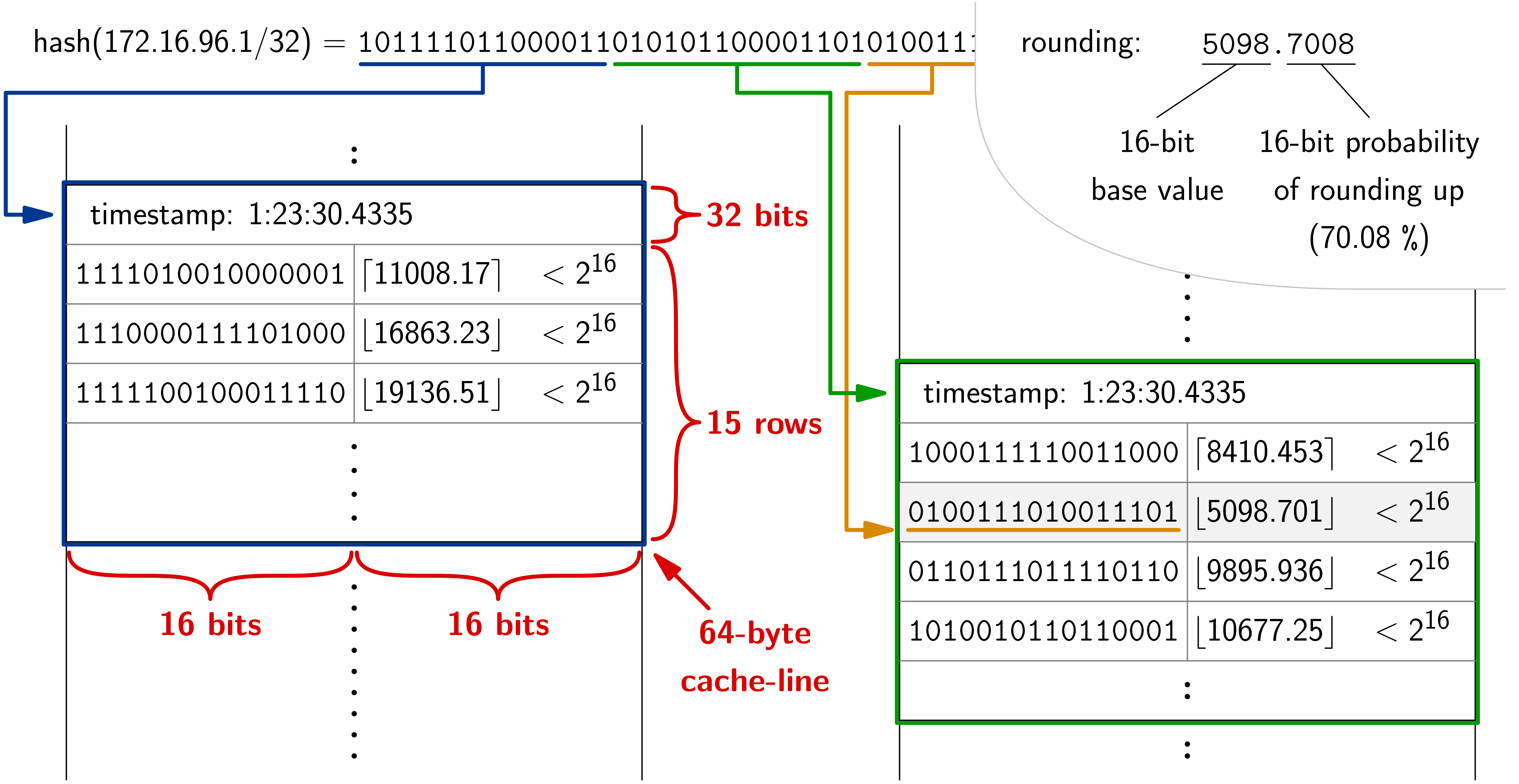
- hashing
  - buckets
  - two tables
- evicting
  - normalized limits
  - choosing minimal
  - keeping value
- lazy decay
- memory layout
  - hashed labels
  - **prob. rounding**



Implementation

- 15x 16-bit label and 16-bit value + 32-bit timestamp is just cache-line.
- Just 2 cache-lines per prefix needed for request, at most 10 in total.

- hashing
  - buckets
  - two tables
- evicting
  - normalized limits
  - choosing minimal
  - keeping value
- lazy decay
- memory layout
  - hashed labels
  - prob. rounding
  - **fit in cache-line**

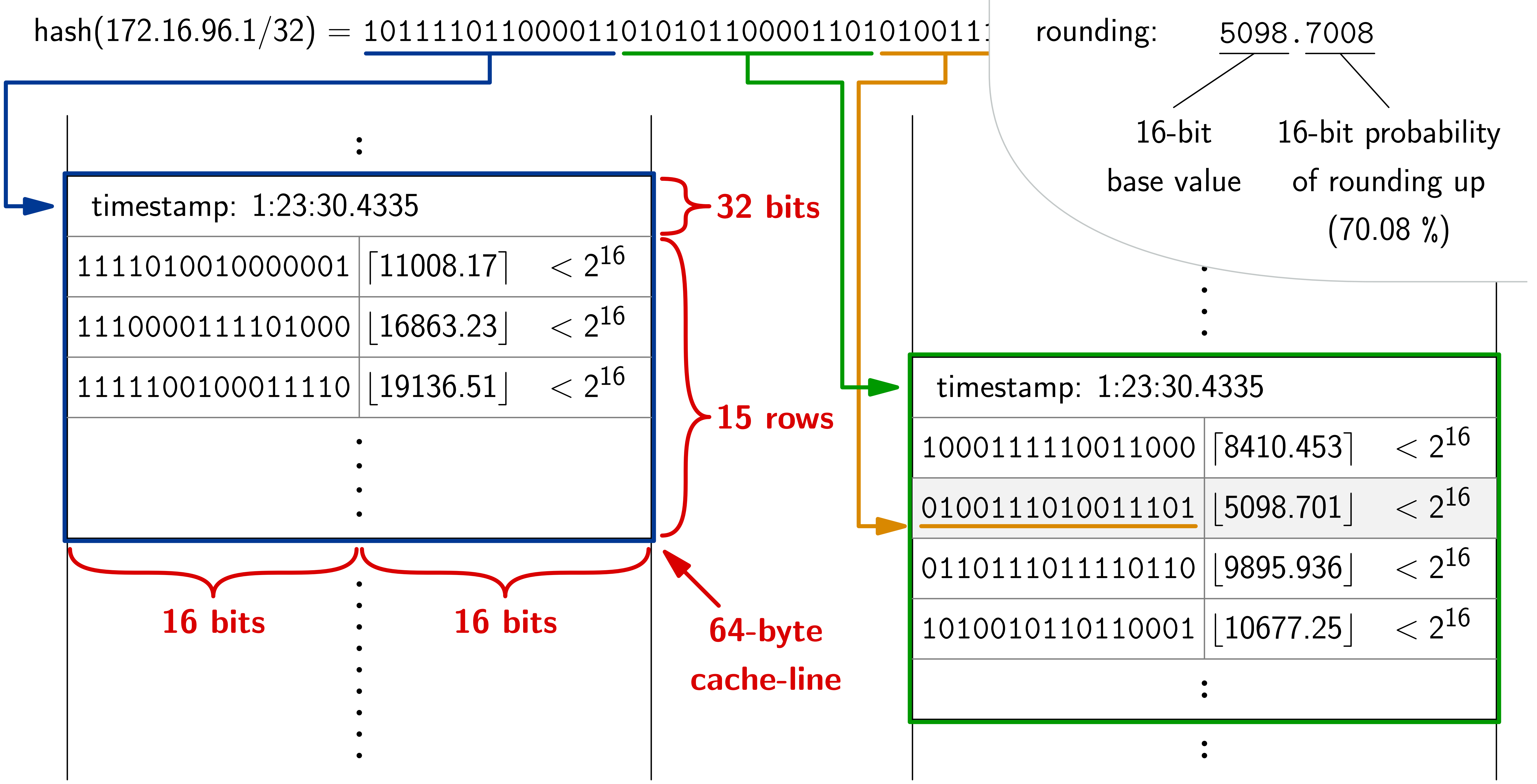


 CZ.NIC CZ DOMAIN REGISTRY

Implementation

- Further optimizations include:
  - Prefetching the 10 cache-lines before accessing.
  - Using atomic instructions to make the data structure lock-free.
  - Using vector instructions to speedup searching in buckets.

- hashing
  - buckets
  - two tables
- evicting
  - normalized limits
  - choosing minimal
  - keeping value
- lazy decay
- memory layout
  - hashed labels
  - prob. rounding
  - fit in cache-line
- **optimizations**
  - **prefetching**
  - **lock-free**
  - **vectorization**



Summary

- rate-limiting
  - counting UDP queries
  - truncating or dropping
- prioritization
  - measuring time
  - reordering
- counters
  - instant/rate limit
  - exponential decay
  - higher limits for shorter prefixes
- implementation⇒