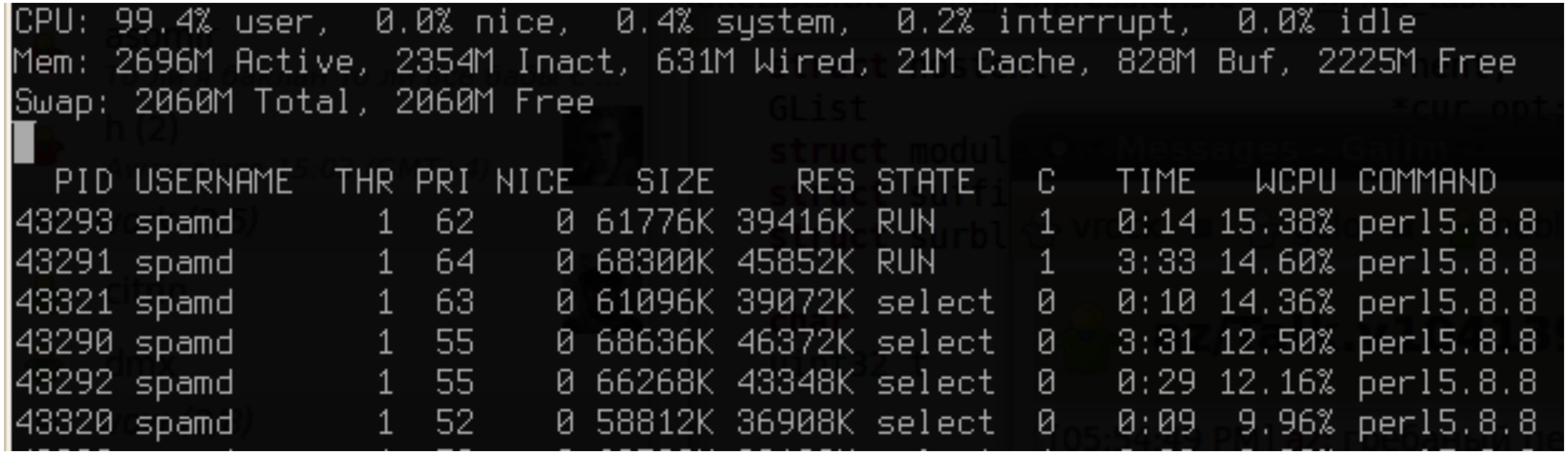
RSPAMD

spam filtering system

Vsevolod Stakhov
https://rspamd.com

Why rspamd?

A real example



Rspamd in nutshell

- Uses multiple rules to evaluate messages scores
- Is written in C
- Uses event driven processing model
- Supports plugins in LUA
- Has self-contained management web interface

Design goals

- Orientation on the mass mail processing
- Performance is the cornerstone of the whole project
- State-of-art techniques to filter spam
- Prefer dynamic filters (statistics, hashes, DNS lists and so on) to static ones (plain regexp)

Part I: Architecture

Event driven processing

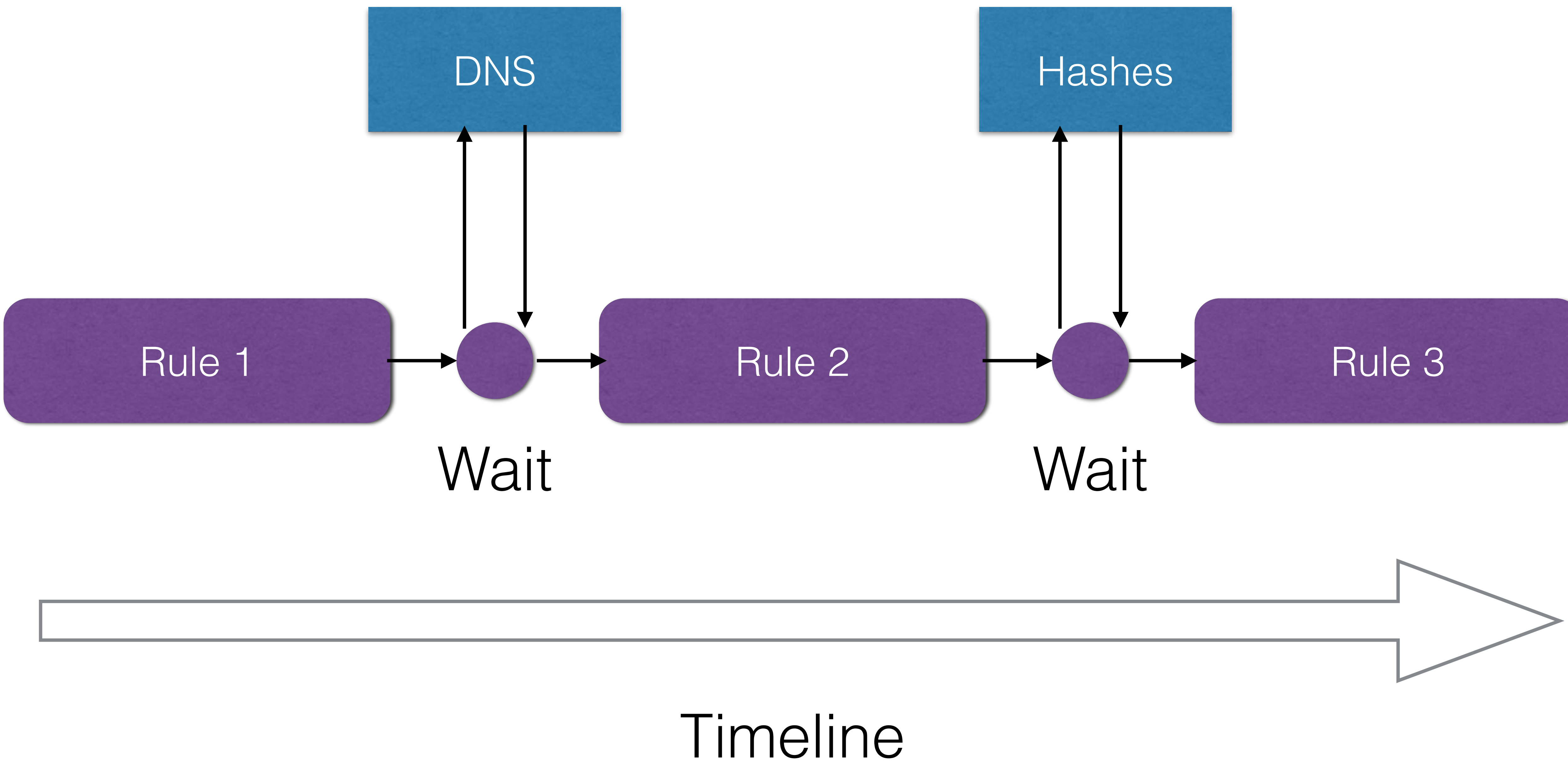
Never blocks*

- Pros:
  - ✅ Can process rules while waiting for network services
  - ✅ Can send all network requests simultaneously
  - ✅ Can handle multiple messages within the same process
- Cons:
  - 📛 Callbacks hell (hard development)
  - ⛔ Hard to limit memory usage due to unlimited concurrency

*almost all the time

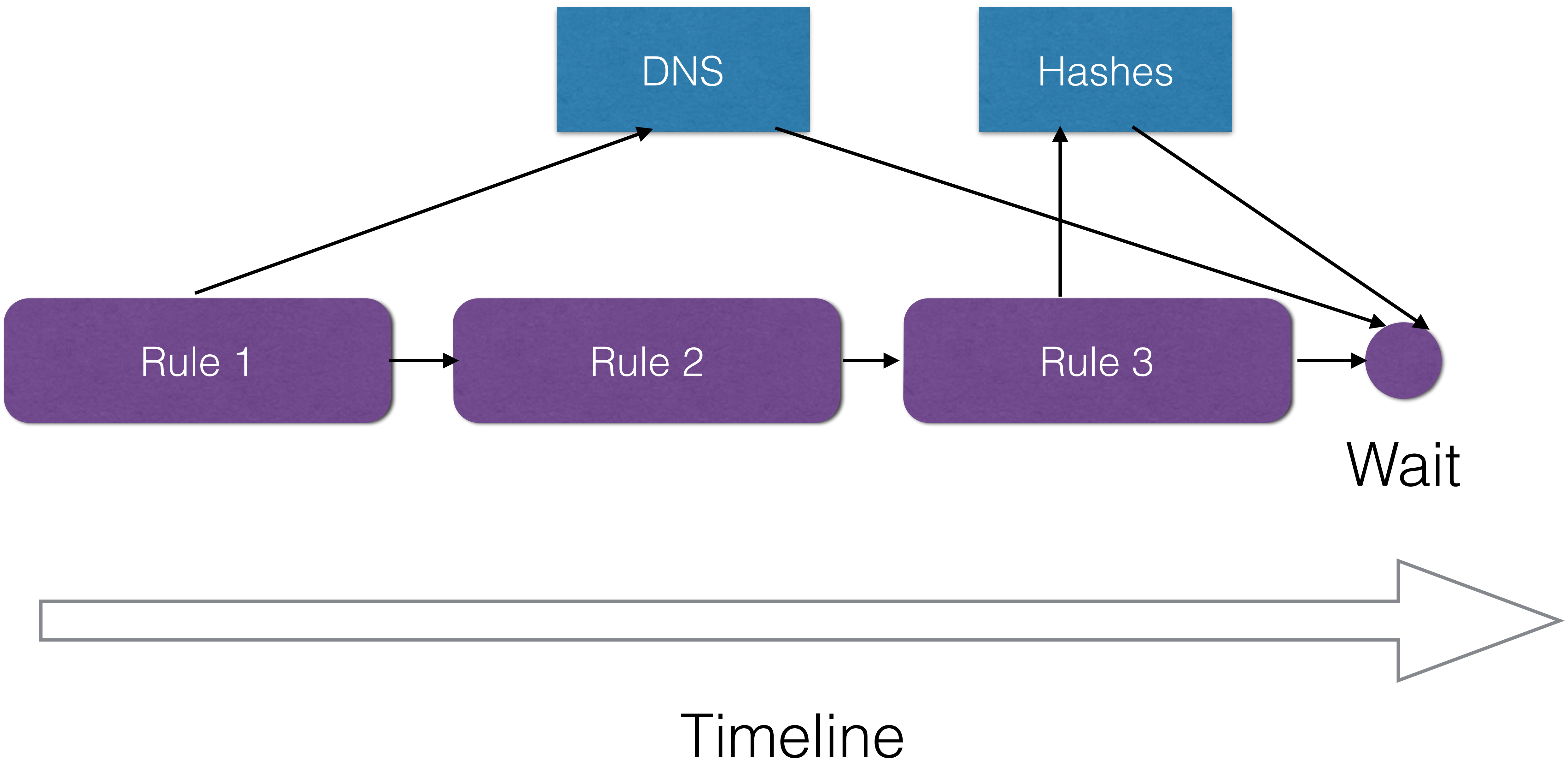
Sequential processing

Traditional approach



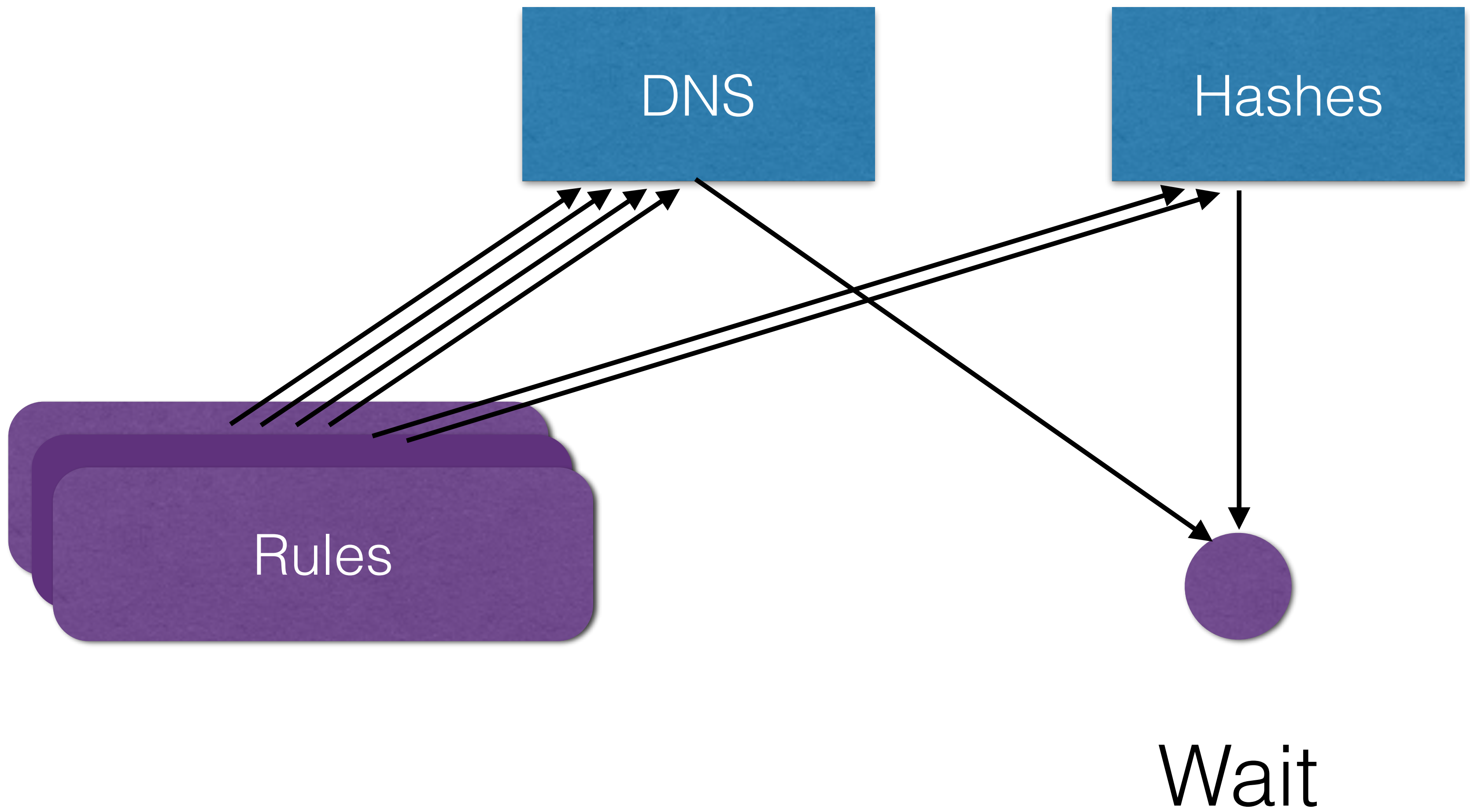
Event driven model

Rspamd approach



Event driven model

What happens in the real life



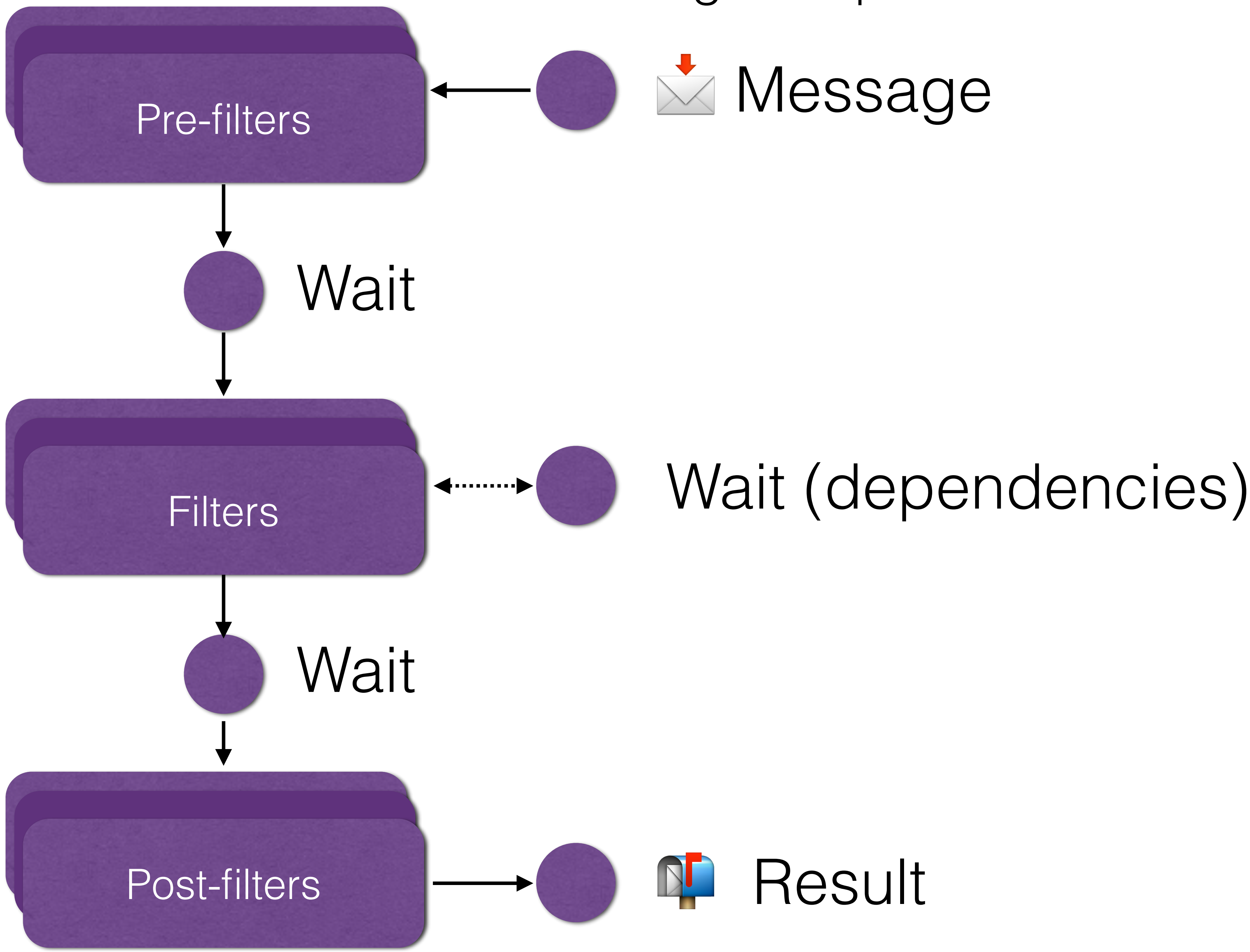
Event driven model

Some measurements

- Rspamd can send hundred thousands of DNS requests per second (RBL, URI blacklists, custom DNS lists): time: 5540.8ms real, 2427.4ms virtual, dns req: 120543
- For small messages (which are 99% of typical mail) network processing is hundreds times more expensive than direct processing: time: 996.140ms real, 22.000ms virtual,
- Event model scales very well allowing highest possible concurrency level within a single process (no locking is needed normally)

Real message processing

We need to go deeper



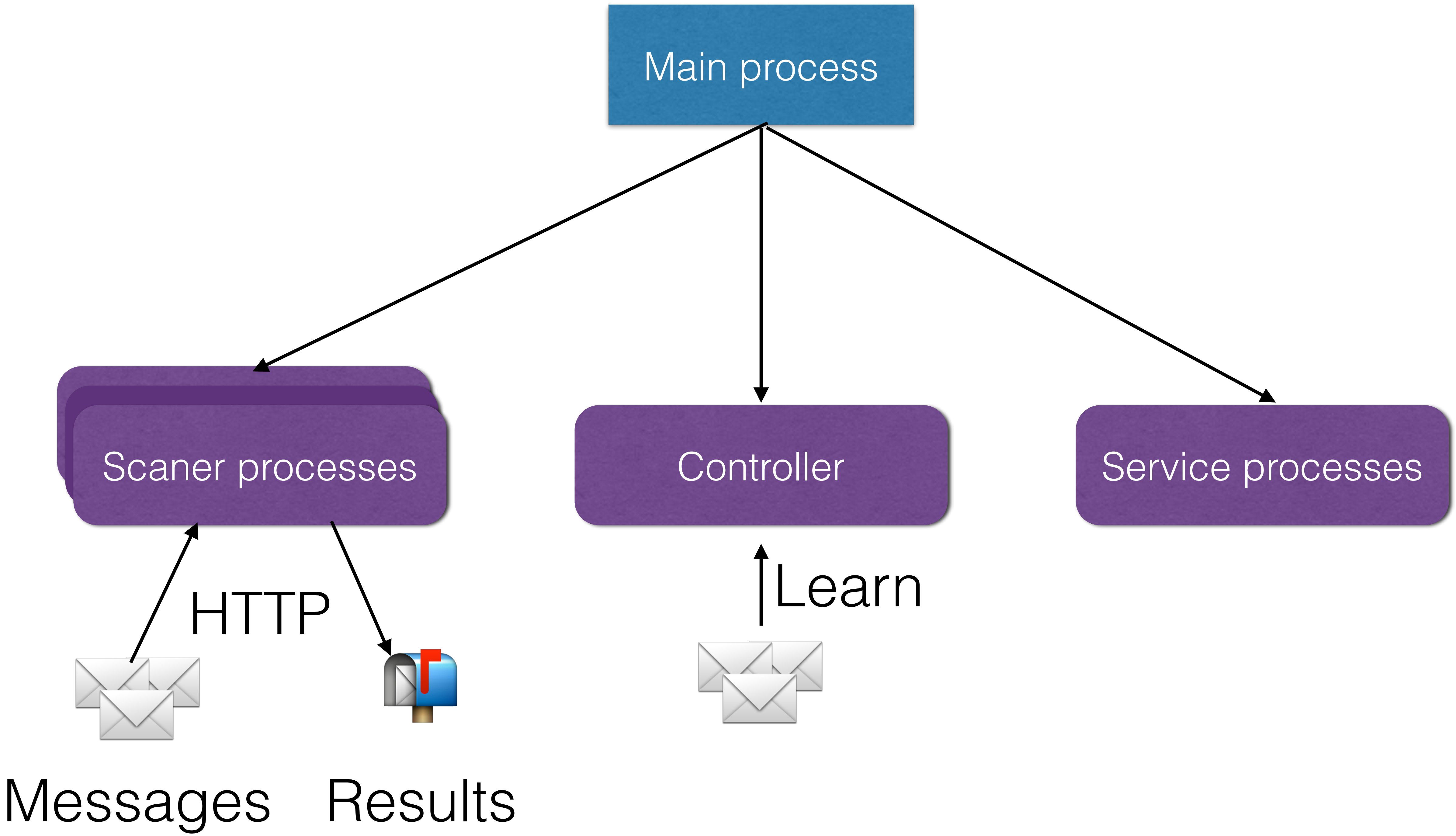
Real message processing

We need to go deeper

- **Pre filters** are used to evaluate message or to reject/accept it early (e.g. greylisting)
- **Normal rules** add scores (positive or negative)
- **Post filters** combine rules and adjust scores if needed (e.g. composite rules)
- Normal rules can also depend on each other (additional waiting)

Rspamd processes

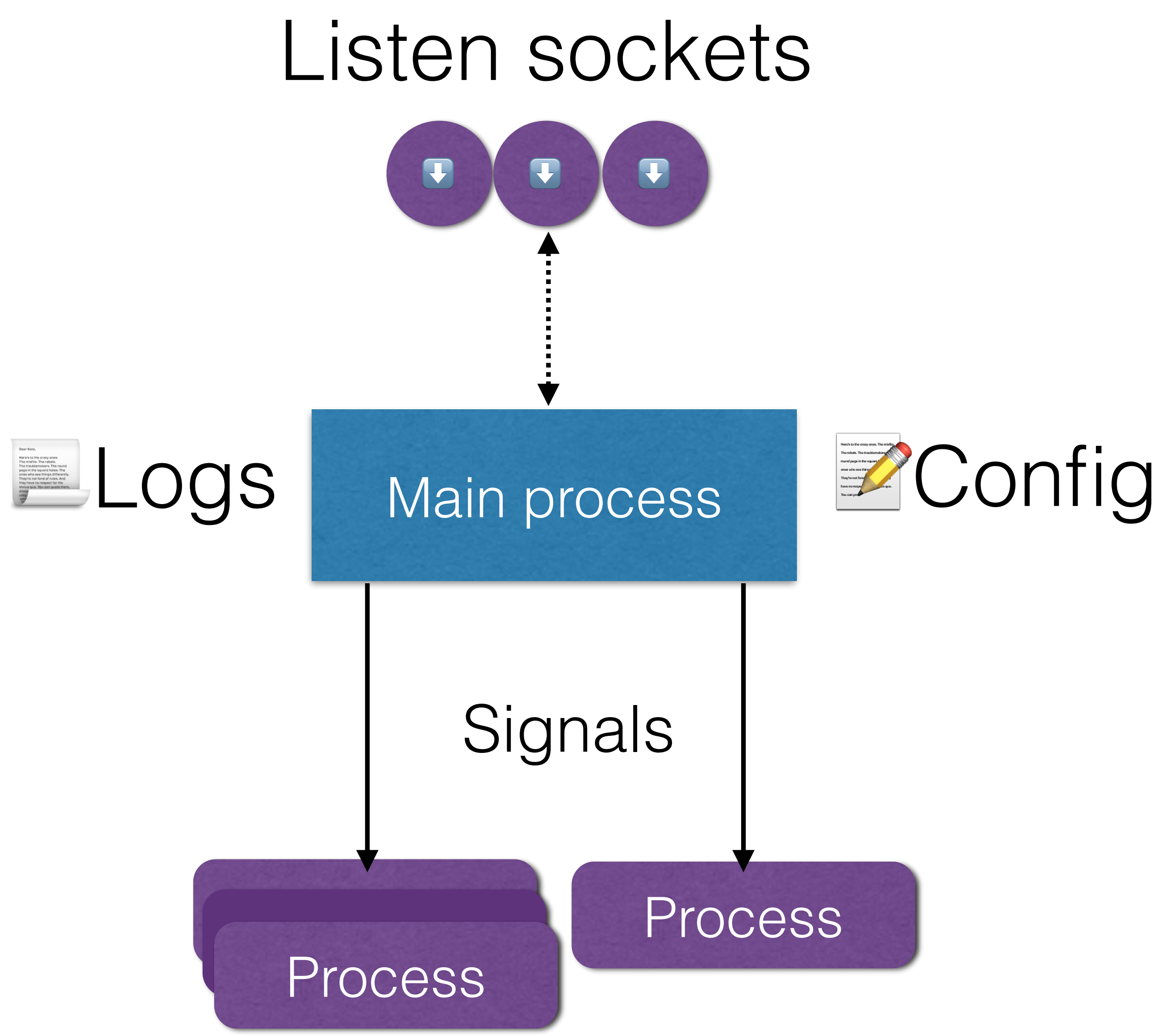
Overview



Main process

One to rule them all…

- Reads configuration
- Manages worker processes
- Listens on sockets
- Opens and reopen log files
- Handles dead workers
- Handles signals
- Reloads configuration
- Handle command line



Scanner process

- Scans messages and returns result
- Uses **HTTP** for operations
- Reply format is **JSON**
- Has **SA** compatibility protocol

Controller worker

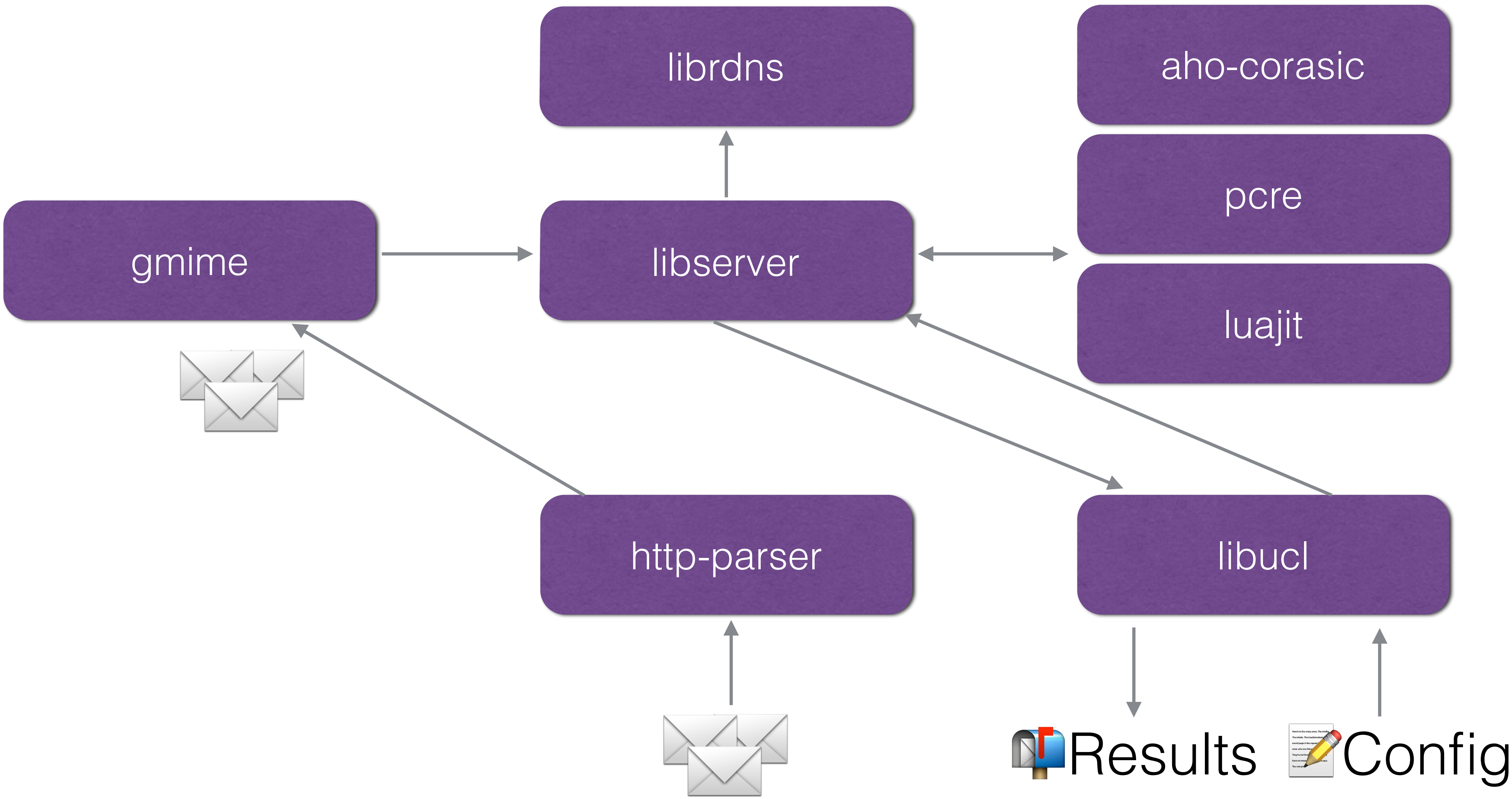
- Provides data for web interface (acts as HTTP server for AJAX requests and serving static files)
- Is used to learn statistics and fuzzy hashes
- Has 3 levels of access:
  - **Trusted IP** addresses (both read and write)
  - **Normal** password* (read commands)
  - **Enable** password* (all commands)

* Passwords are encouraged to be stored encrypted using slow hash function

Service workers

- Are used by rspamd internally and usually have no external API
- The following types are defined:
  - **Fuzzy storage** — stores fuzzy hashes and is learned from the controller and accessed from scanners
  - **Lua worker** — LUA application server
  - **SMTP proxy** — SMTP balancing proxy with RBL filtering
  - **HTTP proxy** — balancing HTTP proxy with encryption support

Internal architecture



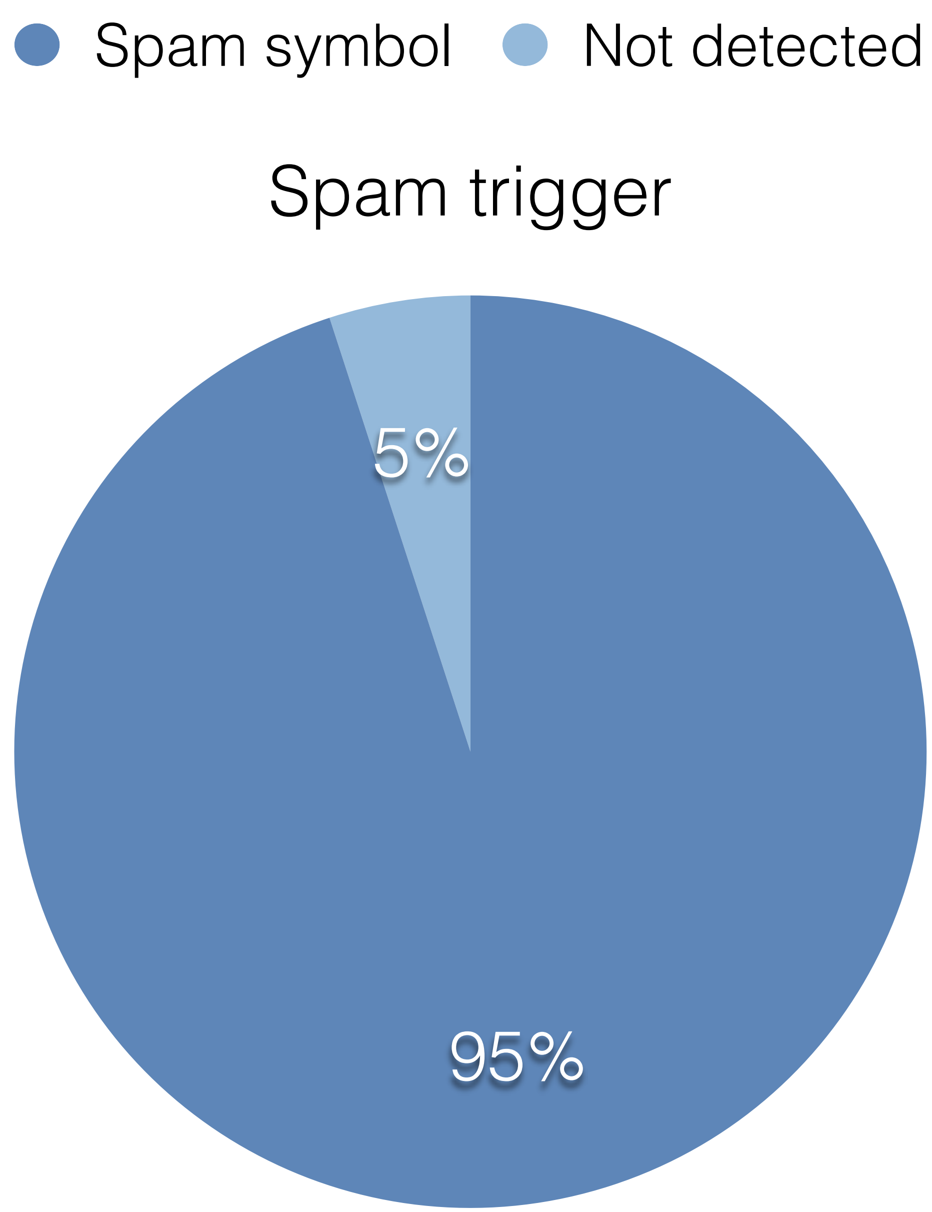
Statistics architecture

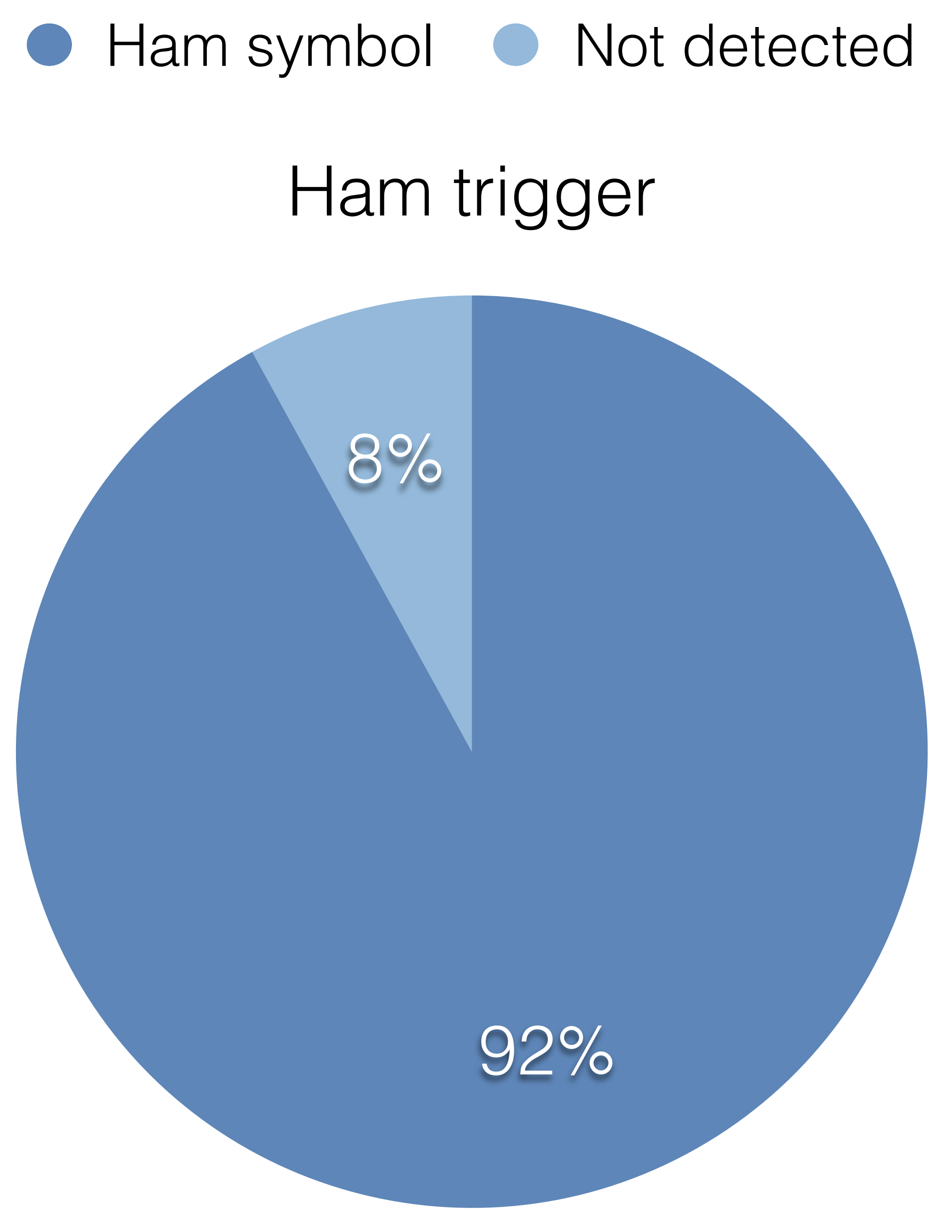
Bayes operations

- Uses sparsed 5-gramms
- Uses messages’ metadata (User-Agent, some specific headers)
- Uses inverse chi-square function to combine probabilities
- Weights of the tokens are based on theirs positions

Statistics benchmarks

Hard cases (images spam)

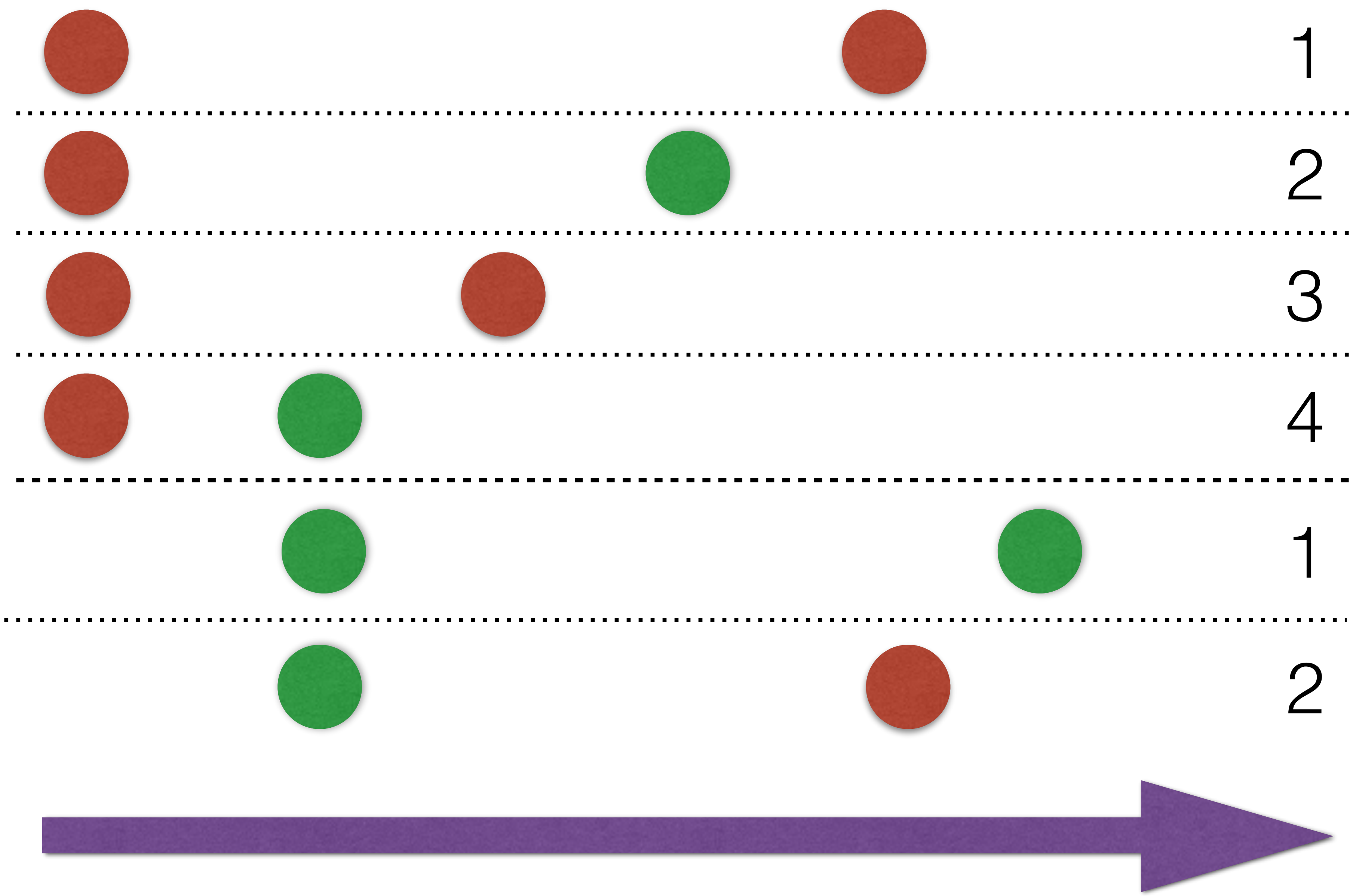




Statistics architecture

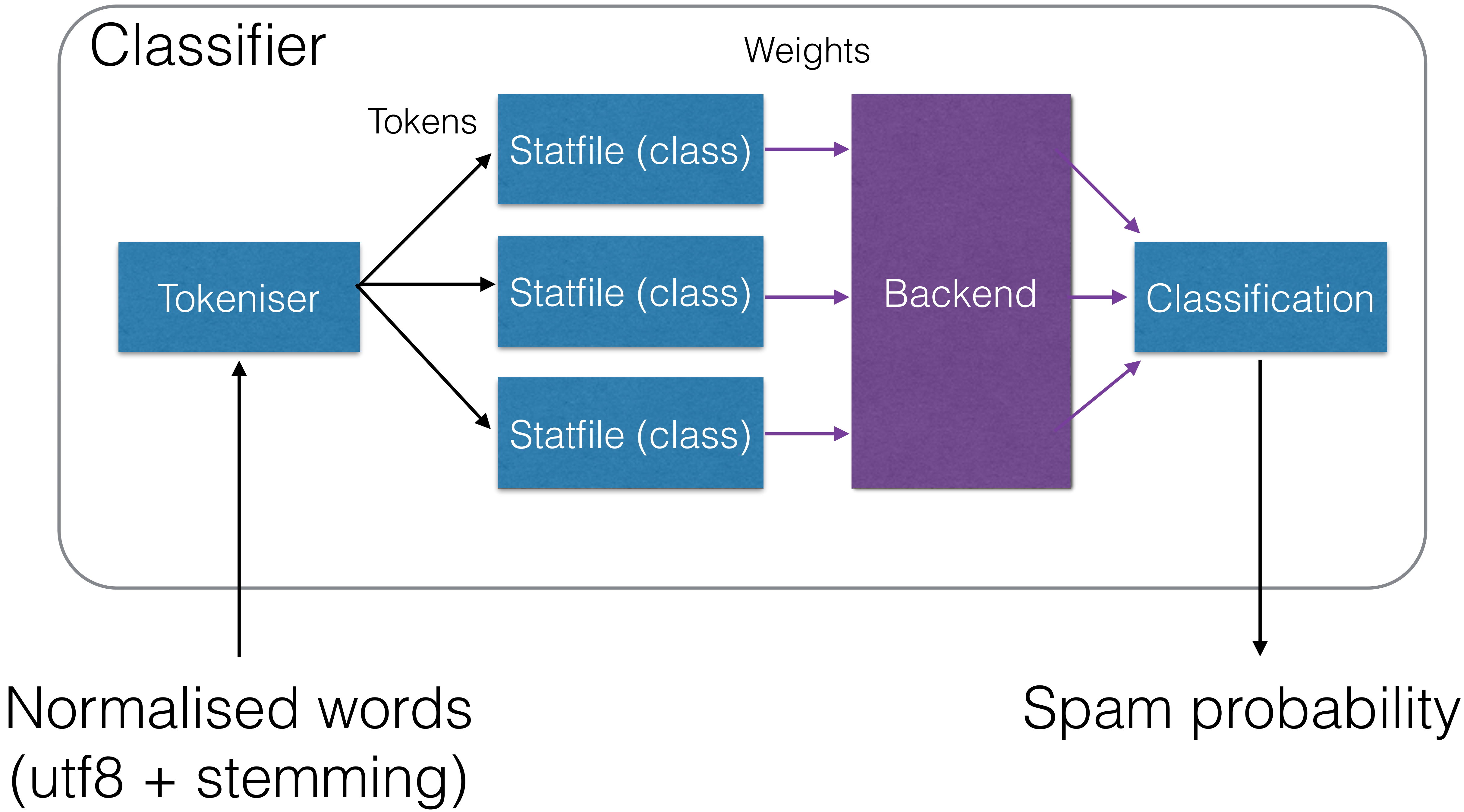
Bayes tokenisation

Quick brown fox jumps over lazy dog



Statistics architecture

Statistics architecture



Fuzzy hashes

Overview

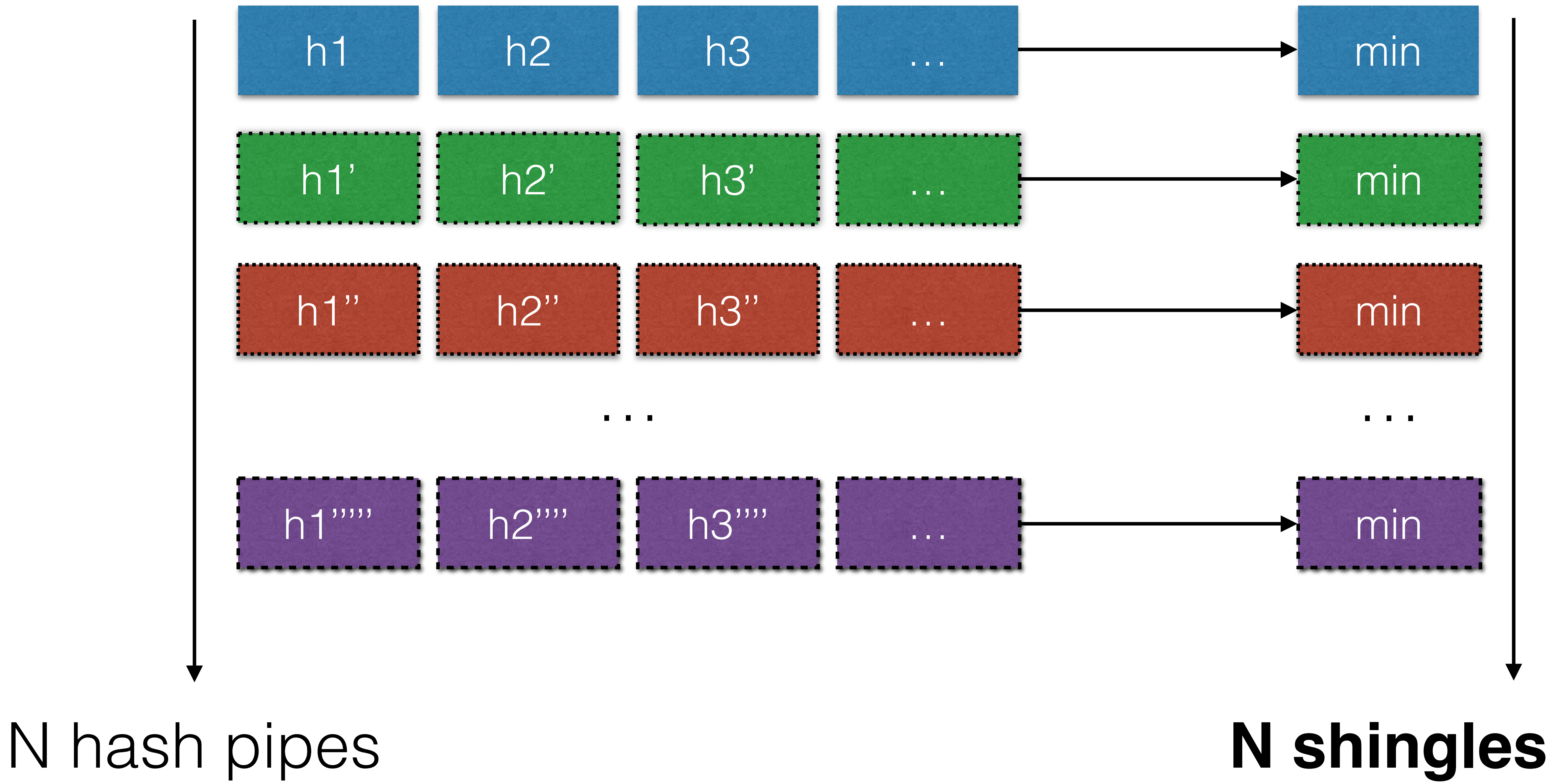
- Are used to match, not to classify a message
- Combine exact hashes (e.g. for images or attachments) with shingles fuzzy match for text
- Use sqlite3 for storage
- Expire hashes slowly
- Write to all storages, read from random one

Fuzzy hashes

Shingles algorithm

Fuzzy hashes

Shingles algorithm



Fuzzy hashes

Shingles algorithm

- Probabilistic algorithm (due to min hash)
- Use sliding window for matching words
- N *siphash* contexts with derived keys
- Derive subkeys using *blake2* function
- Current settings: window size = 3, N = 32

Part II: Performance

Overview

- Rspamd is focused on performance
- No unnecessary rules are executed
- Memory is organised in memory pools
- All performance critical tasks are done by specialised finite-state-machines
- Approximate match is performed if possible

Rules optimisation

Global optimisations

- **Stop** processing when rejection score is hit
- Process **negative** rules first to avoid FP errors
- Execute **less expensive** rules first:
    - Evaluate rules average execution time, score and frequency
    - Apply greedy algorithm to reorder
    - Resort periodically

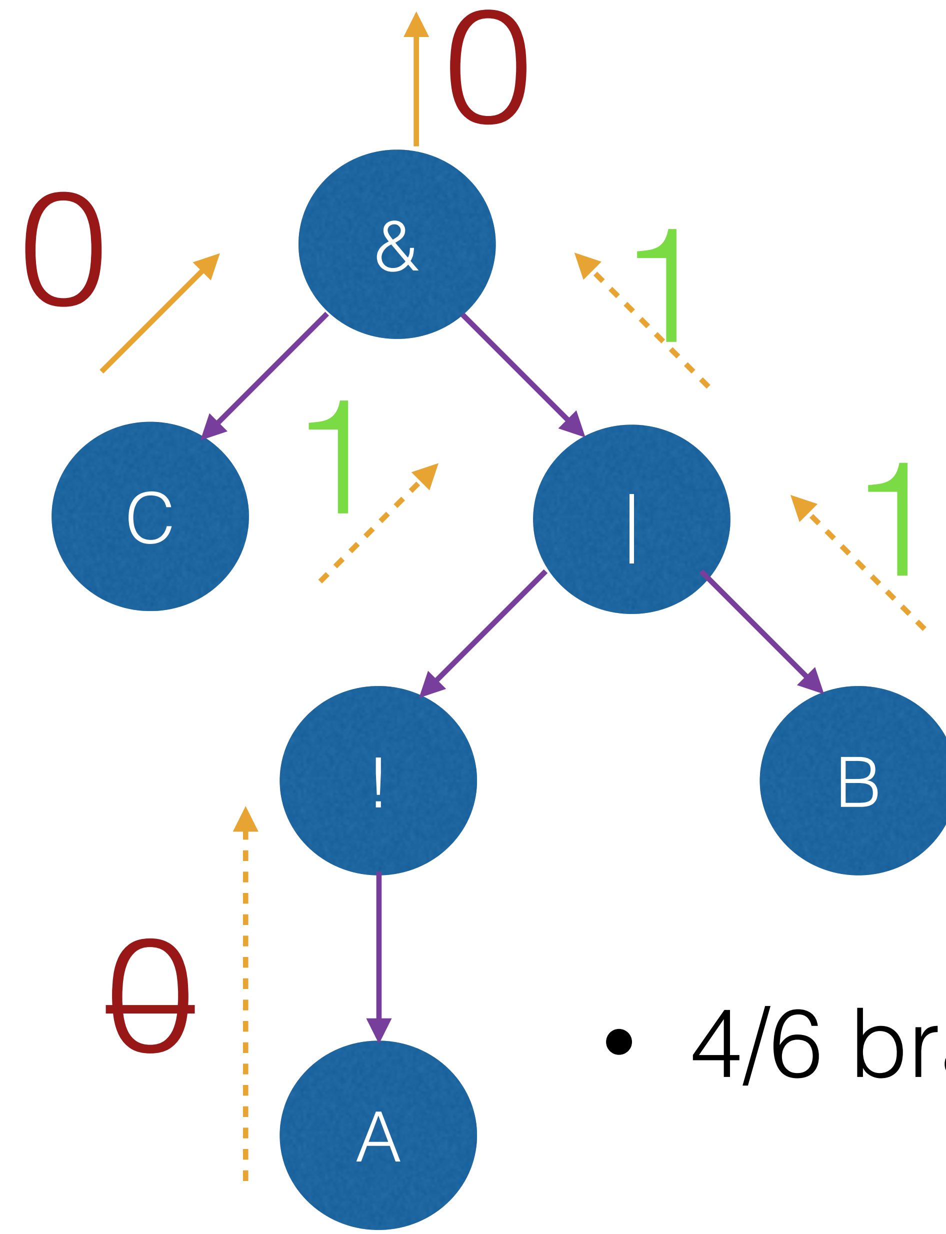
Rules optimisation

Local optimisations

- Each rule is additional optimised using abstract syntax tree (**AST**): 3-4 times speed up for large messages
- Each rule is split and reordered using the similar greedy algorithm
- Regular expressions are compiled using **PCRE JIT** (from 50% to 150% speed up usually)
- Lua is optimised using **LuaJIT**

AST optimisations

Branches cut



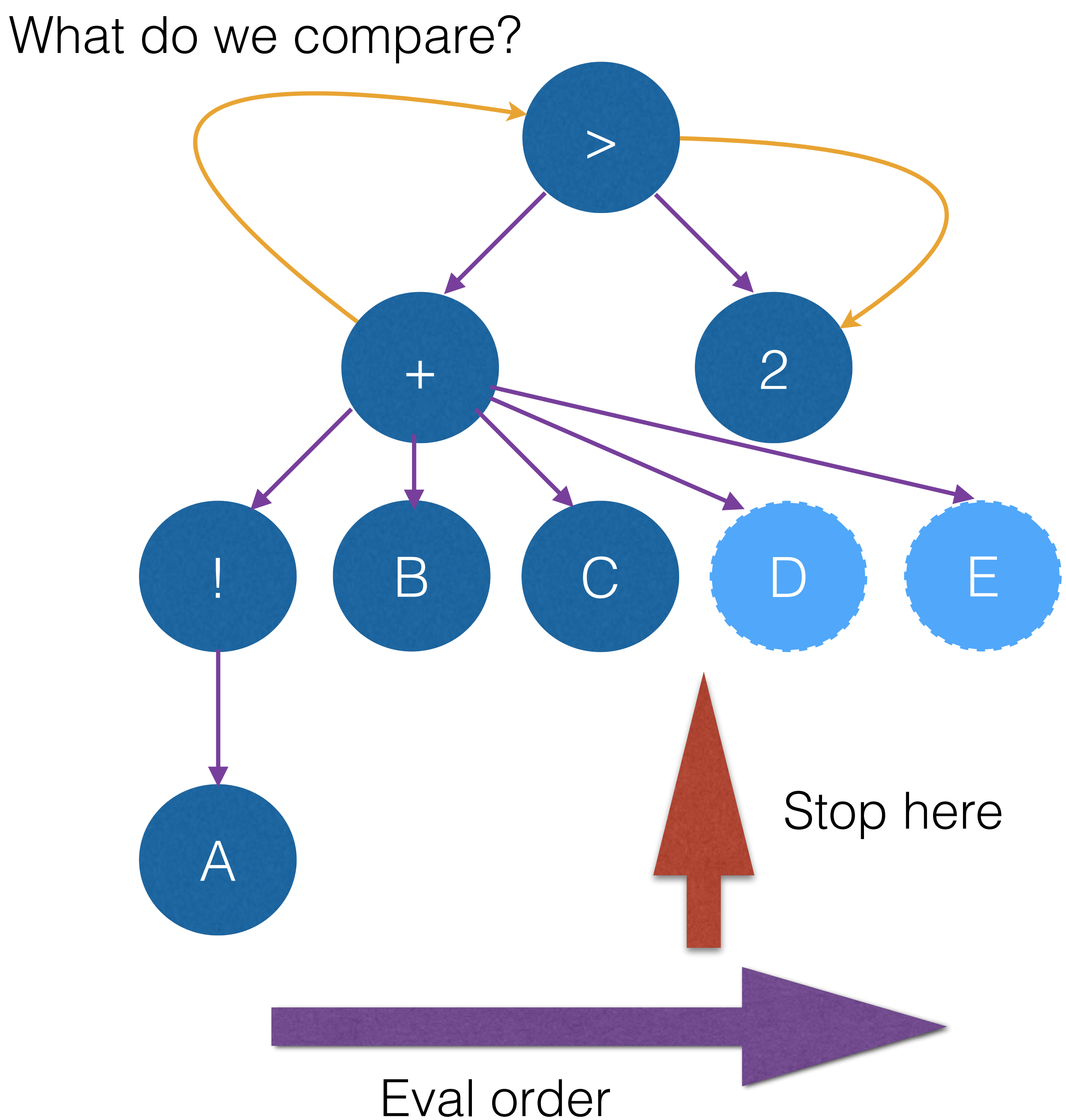
- 4/6 branches skipped

A = 0, B = 1, C = 0

Eval order

AST optimisations

N-ary optimisations

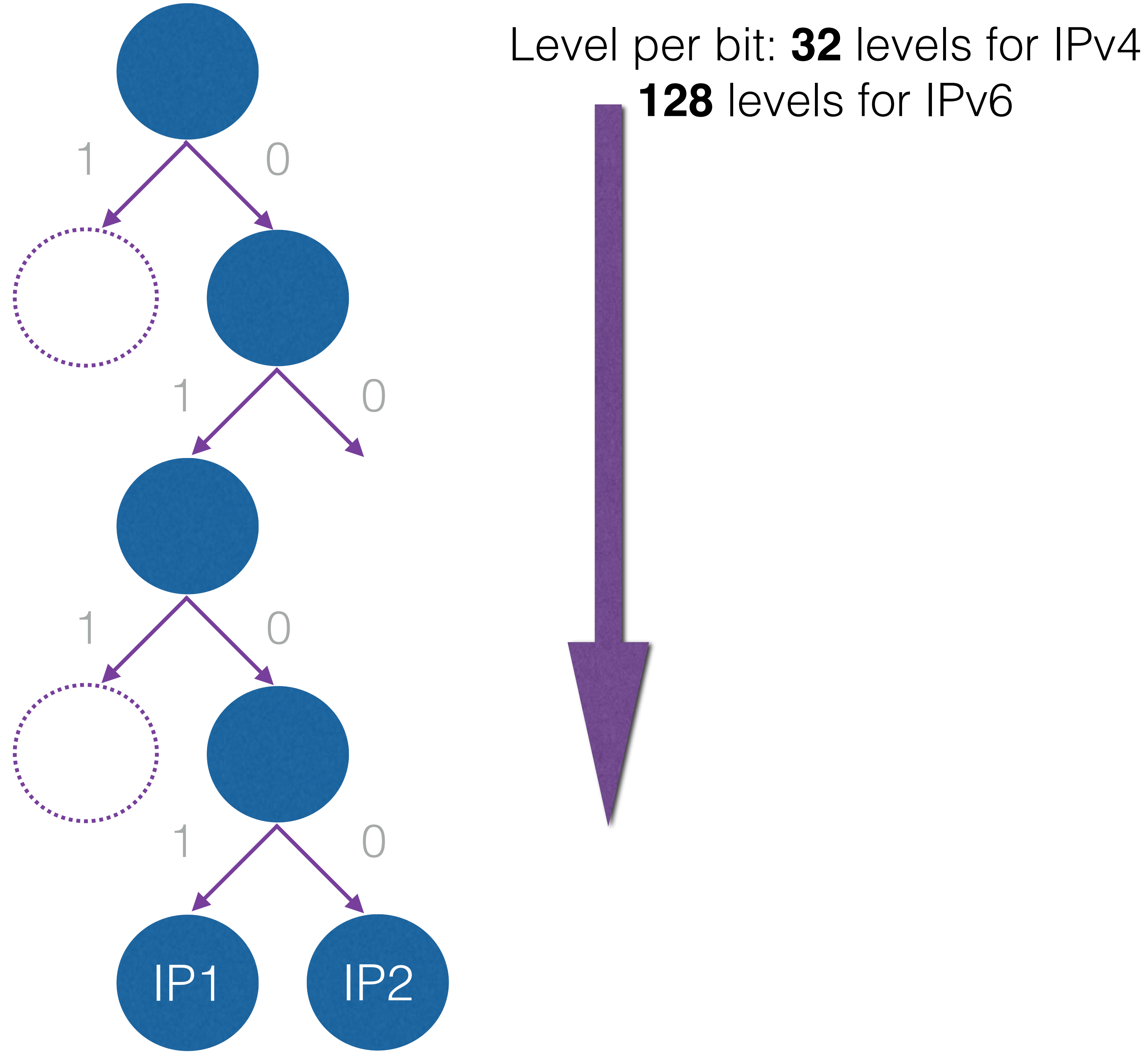


Parsing FSM

- For the most of time consuming operations, rspamd uses special finite-state machines:
  - headers parsing;
  - received headers parsing;
  - protocol parsing;
  - URI parsing;
  - HTML parsing
- Prefer approximate matching, meaning extraction of the most important information and skipping less important details

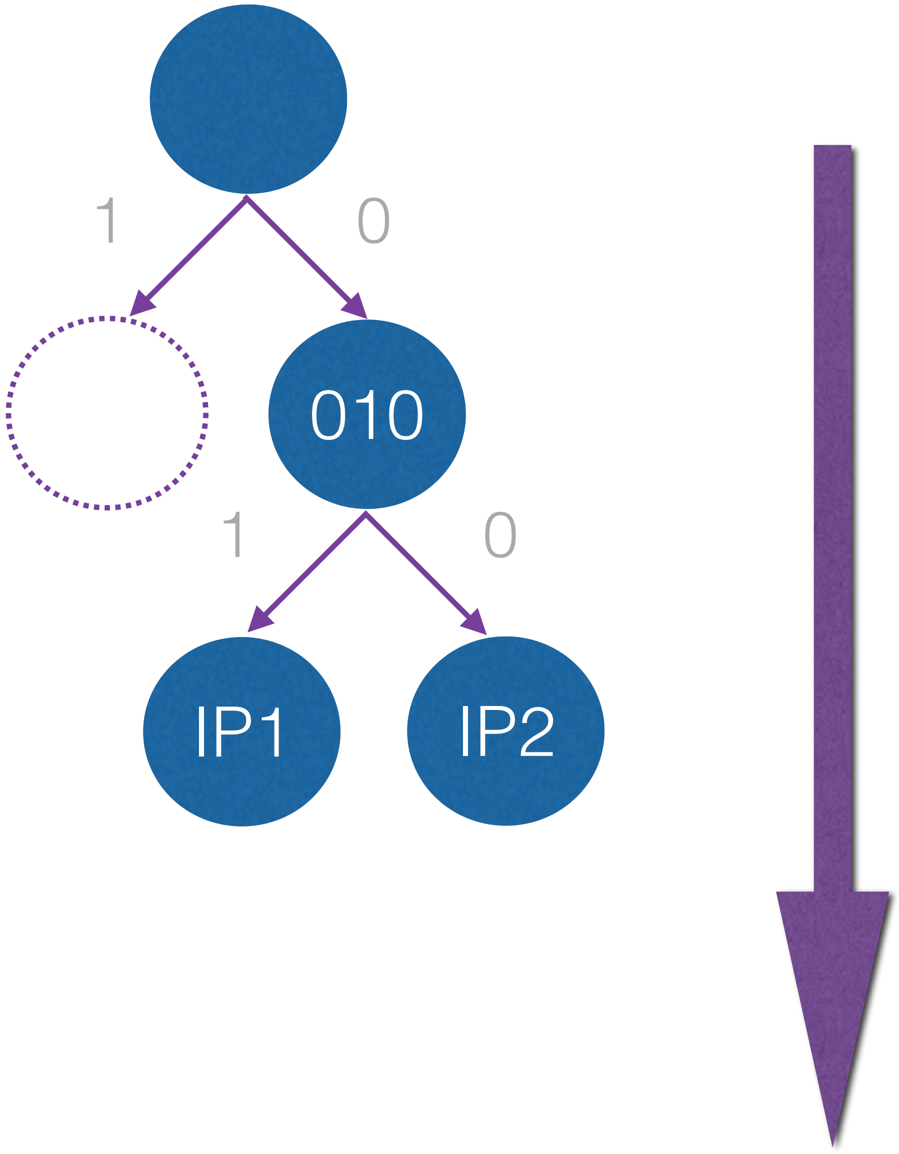
IP addresses storage

Traditional radix trie



IP addresses storage

Prefix skipped radix trie



IP addresses storage

Prefix skipped radix trie

- Can efficiently compress IP prefixes
- Lookup is much faster due to lower trie depth
- IPv4 and IPv6 addresses can live within a single trie
- Insertion is also faster
- Algorithm is much harder but extensively tested

Library optimisations

Logger interface

- Universal logger for files/syslog/console
- Filters non-ascii (or non-utf8 if enabled) symbols
- Allows skipping of repeated messages
- Can disable processing in case of throttling
- Can handle both privileged and non-privileged reopening

Library optimisations

Printf interface

- Libc printf is slow and stupid
- Rspamd printf is inspired by **nginx** printf:
  - Supports fixed integers (int64_t, uint32_t)
  - Supports fixed length string (*%v*)
  - Supports encoded strings and numbers (human-readable, hex encoding, base64 and so on)
  - Supports various backends: fixed size buffers, automatically growing strings, files, console…
- Rspamd *printf* **does not** try to print input when output is overflowed (so it's impossible to force it to use CPU resources for ridiculously large strings)

Library optimisations

String operations

- Fast base64/base32 operations:
  - **alignment** optimisations;
  - use loop **unwinding**;
  - use **64 bit** integers instead of characters
- Fast lowercase:
  - use the same optimisations for ASCII string
  - approximate lowercase for UTF8 (not 100% correct but much faster)
- Fast **lines counting**: http://git.io/vYIdq

Library optimisations

Generic tools

- Fast **hash** functions (*xxhash* and *blake2*)
- Fast **encryption** (using SIMD instructions if possible)
- Use ***mmap*** when possible
- **Align** memory for faster operations
- Use google performance tools to find bottlenecks

Part III: Security

Main points

- Maintaining secure coding is hard for C:
  - Prefer **fixed length** strings
  - **Avoid** insecure functions
  - **Abort** if malloc fails
  - **Assertions** on bad input
  - Testing (functional + unit testing)
- Main treats:
  - Interaction with **DNS**
  - Passive **snooping** of traffic
  - Specially crafted messages

DNS security

- DNS is the major point to interact with the external world
- There could be thousands requests per second
- DNS replies can be untrusted
- SPF records could be recursive
- DKIM records could be malformed
- Need local and global DNS requests limit

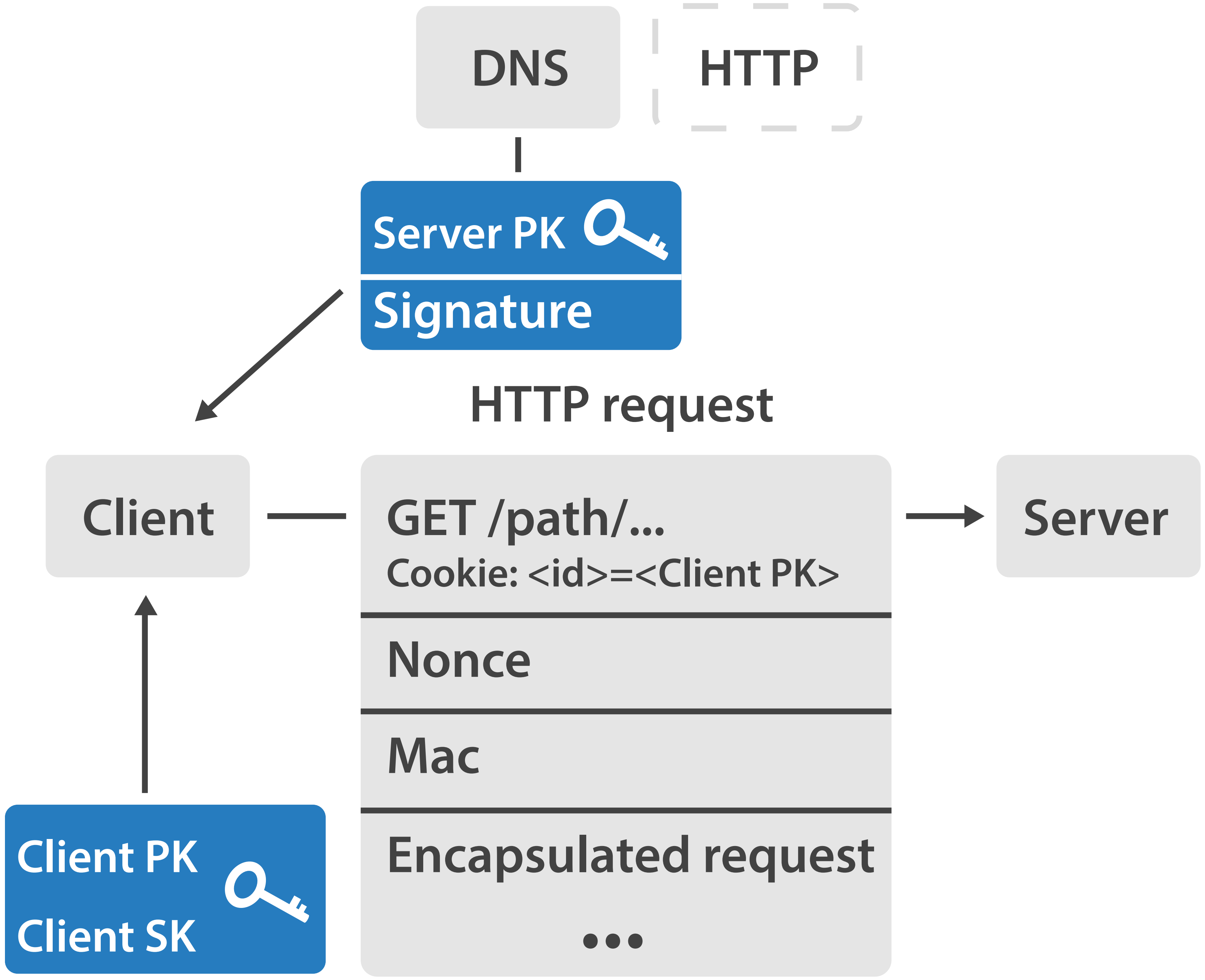
RDNS library

- Uses secure DNS ID generator based on crypto permutation and entropy reseeding
- Uses sockets pool with time/usage expiration
- Randomises source port
- Carefully filters input data (+IDN encoding)

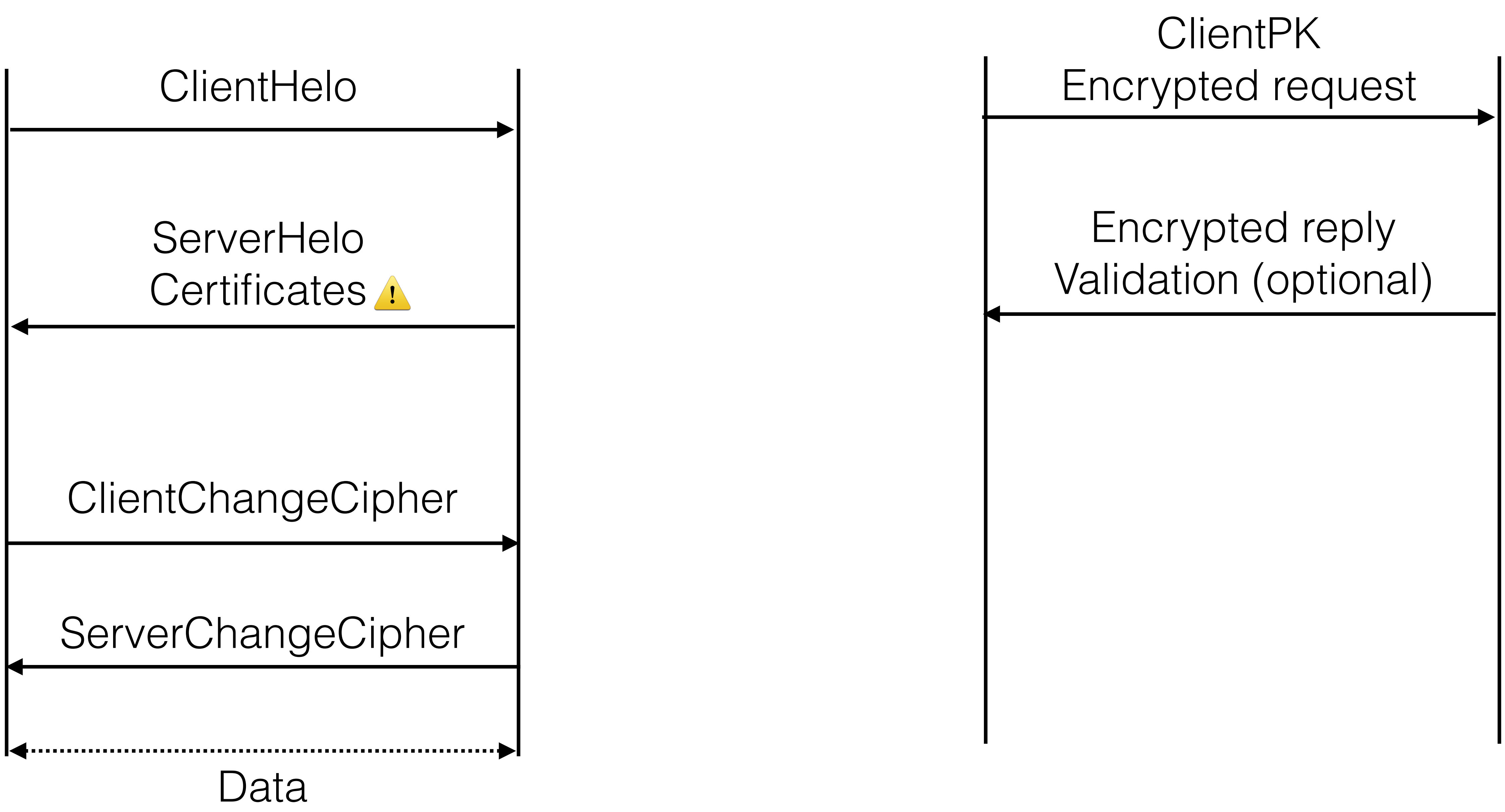
Transport encryption

- Designed to be fast, simple and secure
- TLS is too hard to manage in events based model
- Many functions of TLS are useless for rspamd
- TLS involves intermediate copying and significant latency increase

HTTPCrypt in nutshell



Handshakes

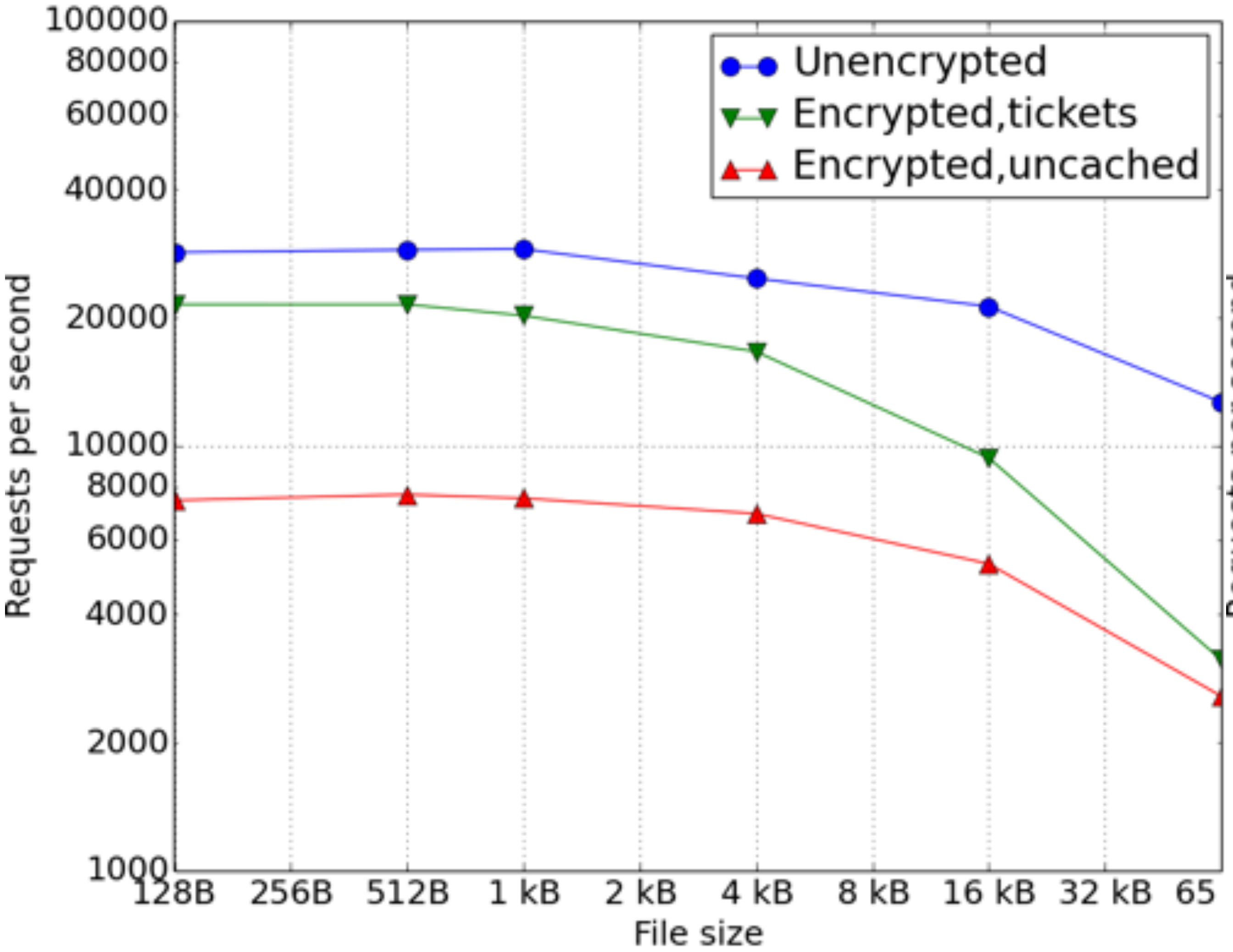


TLS handshake

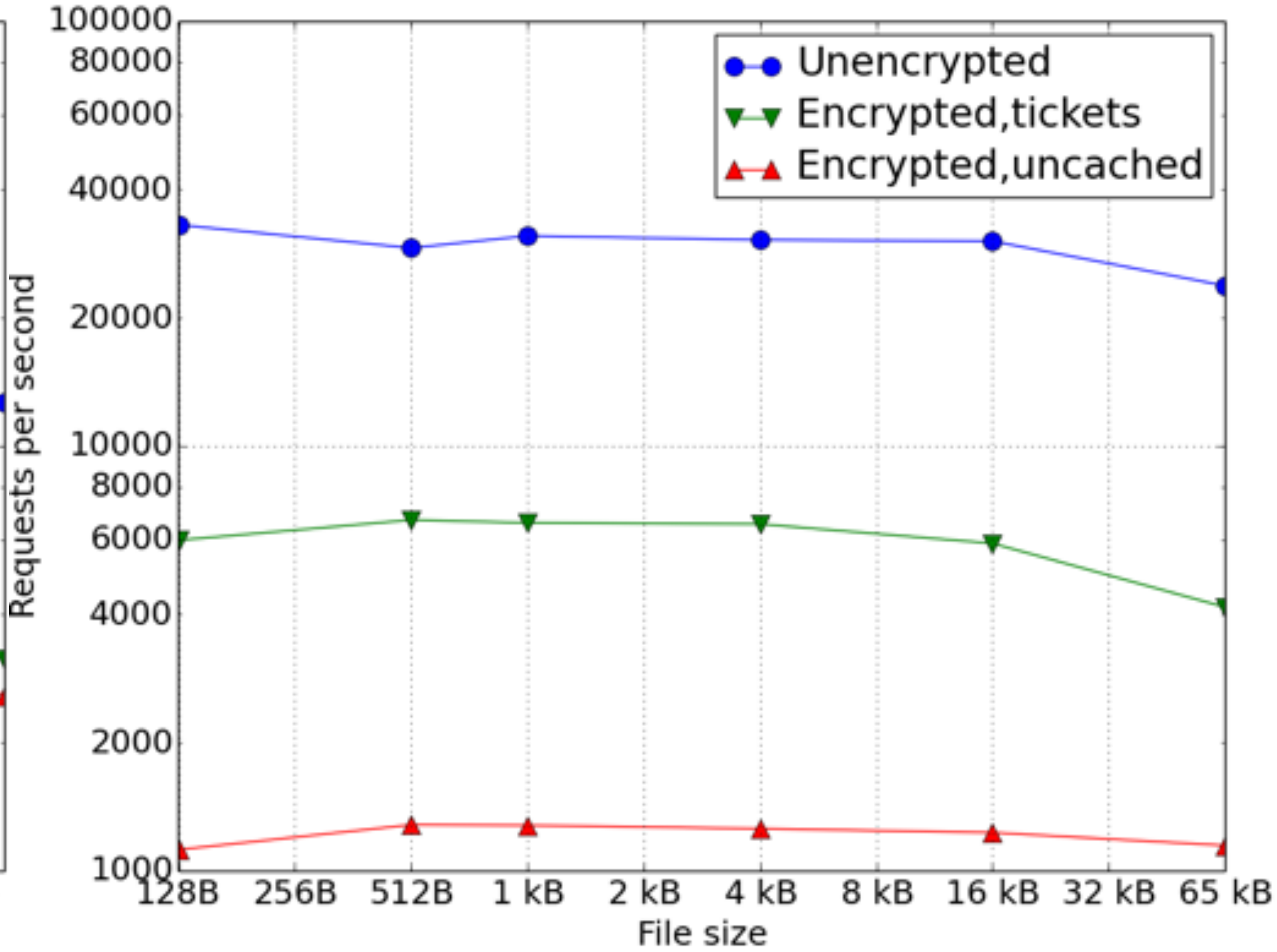
HTTPCrypt handshake

Performance

Throughput



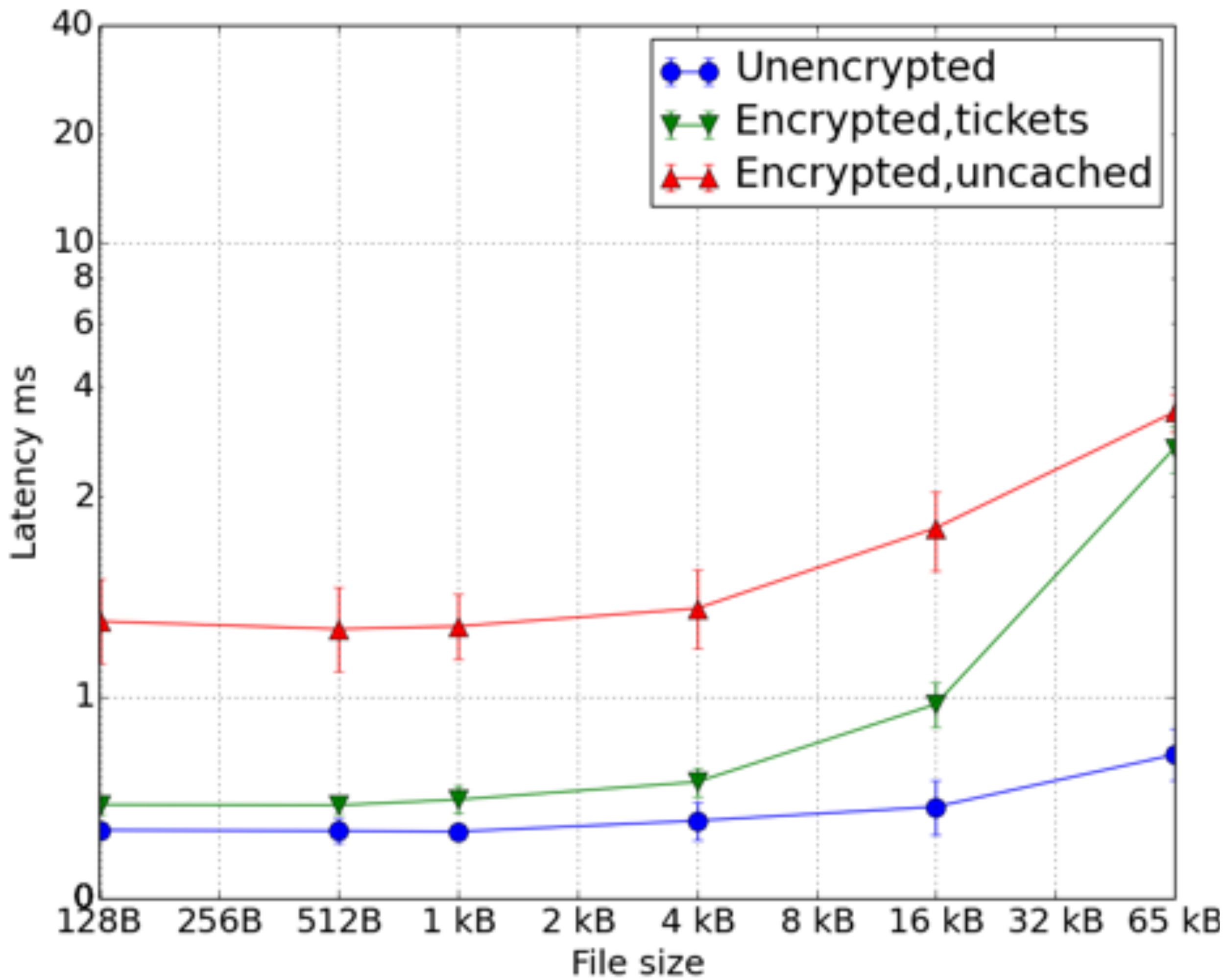
HTTPCrypt



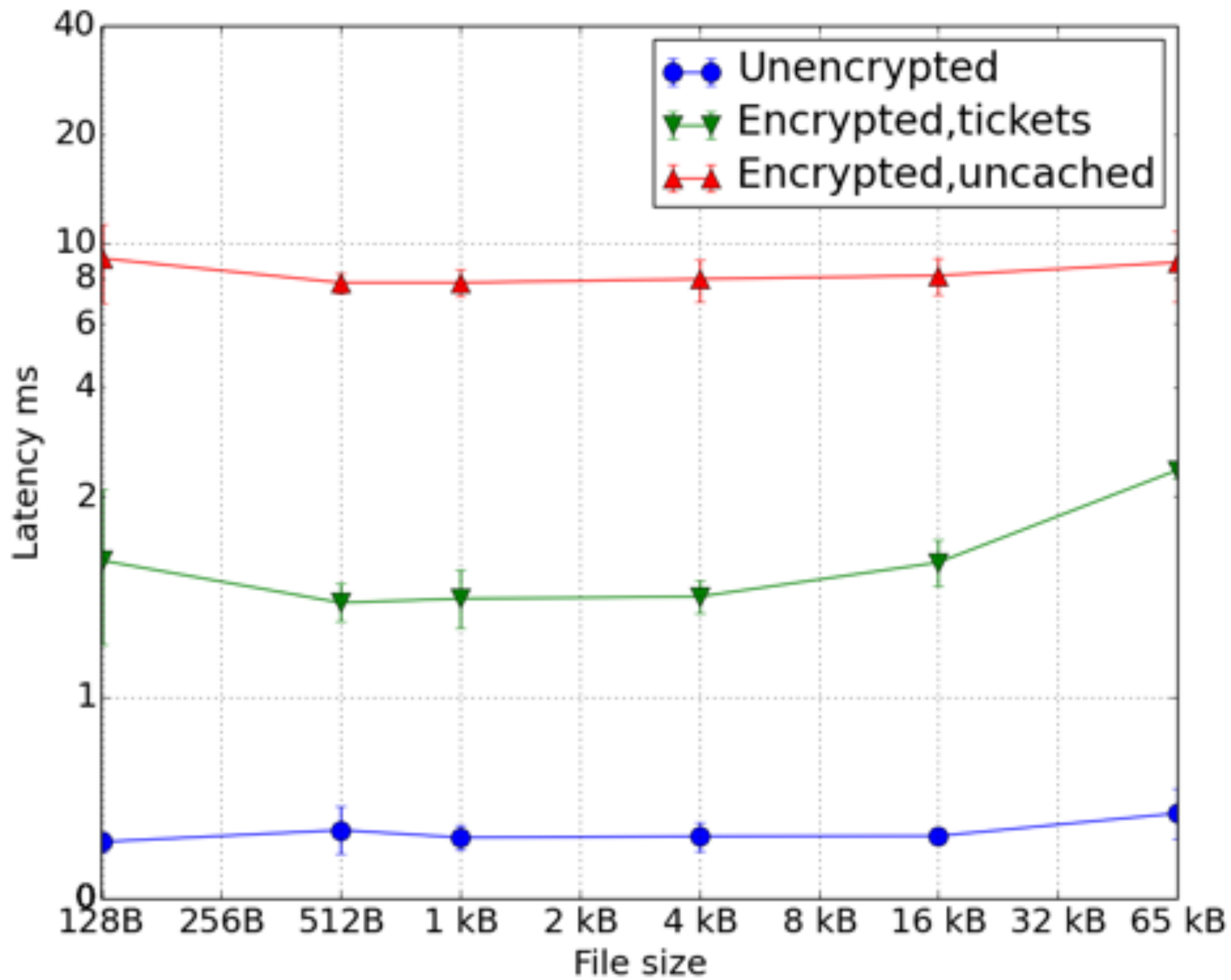
HTTP+TLS (nginx)

Performance

Latency



HTTPCrypt



HTTP+TLS (nginx)

Performance analysis

Why HTTPCrypt is fast

- For new sessions, HTTPCrypt uses curve25519 ECDH which is almost twice faster than NIST P-256 ECDH
- There is no signing operation and no ECDSA
- For bulk encryption, there is no intermediate buffering like in TLS - the payload is encrypted in-place
- Latency is reduced by skipping the full TLS handshake
- Large requests are somehow slow due to lack of chunked encoding in HTTPCrypt implementation and some clever tricks of data reading

Hashes security

- Hash tables are vulnerable for untrusted data:
  - Rspamd randomly chooses hash tables seed at start that is hard to predict
  - *XXHash* is used for good speed and hash distribution
  - *Siphash* is used for public hash tables (e.g. fuzzy hashes)
  - It’s hard to predict hash seed, hence it’s hard to organise computational attack on hash tables

Part IV: Configuration

Configuration evolution

1. Grammar parser (lex + yacc)
    - ⛔ Hard to manage
    - ⛔ Hard to extend
2. XML
    - ⛔ Unreadable
    - ⛔ Problems with expressions (A > B)
3. UCL - universal configuration language
    - ✅ Easy to manage (looks like nginx.conf)
    - ✅ Macro support
    - ✅ JSON data model (can be used as JSON parser)

UCL building blocks

- Sections

```
section {
  key = “value”;
  number = 10K;
}
```

- Arrays

```
upstreams = [
      “localhost:80”,
      “example.com:8080”,
]
```

- Variables

```
static_dir = “${WWWDIR}/“;
filepath = “${CURDIR}/data”;
```

- Macros

```
.include “${CONFDIR}/workers.conf”
.include (glob=true,priority=2) “${CONFDIR}/conf.d/*.conf”
.lua { print(“hey!”); }
```

- Comments

```
key = value; // Single line comment
/* Multiline comment
/* can also be nested */
 */
```

Configuration components

- Each component is normally included to the main configuration
- *rspamd.local.conf* is used to **extend** configuration
- *rspamd.override.conf* is used to **override** values in the configuration
- It is possible to use numeric multipliers: "*k/m/g*" or "*ms/s/m/h/d*" for time values

Global options

Workers configuration

Scores (metrics)

Statistics

Modules configuration

Lua rules

- The most of rules are defined in LUA configuration
- Two types of LUA rules:
    - **Regexp** rules (look like strings)
    - **Lua** functions (pure LUA code)

Lua rules

Some examples

- Regexp rule

```
-- Outlook versions that should be excluded from summary rule
local fmo_excl_o3416 = 'X-Mailer=/^Microsoft Outlook, Build 10.0.3416$/H'
local fmo_excl_oe3790 = 'X-Mailer=/^Microsoft Outlook Express 6.00.3790.3959$/H'
-- Summary rule for forged outlook
reconf['FORGED_MUA_OUTLOOK'] = string.format('(%s | %s) & !%s & !%s & !%s',
                           forged_oe, forged_outlook_dollars, fmo_excl_o3416, fmo_excl_oe3790, vista_msgid)
```

- Lua rule

```
rspamd_config.R_EMPTY_IMAGE = function(task)
  local tp = task:get_text_parts() -- get text parts in a message

  for _,p in ipairs(tp) do -- iterate over text parts array using `ipairs`
    if p:is_html() then -- if the current part is html part
      local hc = p:get_html() -- we get HTML context
      local len = p:get_length() -- and part's length

      if len < 50 then -- if we have a part that has less than 50 bytes of text
        local images = hc:get_images() -- then we check for HTML images

        if images then -- if there are images
          for _,i in ipairs(images) do -- then iterate over images in the part
            if i['height'] + i['width'] >= 400 then -- if we have a large image
              return true -- add symbol
            end
          end
        end
      end
    end
  end
end
```

Pure LUA functions

Review

- Are very powerful
- Have access to all information from rspamd via lua API: https://rspamd.com/doc/lua/
- Are very fast since C <-> LUA interaction is cheap
- Can use zero-copy objects called *rspamd{text}* to avoid copying when moving data between C and LUA

Pure LUA functions

- Variables:

```
local ret = false -- Generic variable
local rules = {} -- Empty table
local rspamd_logger = require “rspamd_logger" -- Load rspamd module
```

- Conditionals:

```
if not ret then -- can use ‘not’, ‘and’, ‘or’ here
…
elseif ret ~= 10 then -- note ~= for ‘not equal’ operator
end
```

- Loops:

```
for k,m in pairs(opts) do … end -- Iterate over keyed table a[‘key’] = value
for _,i in ipairs(images) do … end -- Iterate over array table a[1] = value
for i=1,10 do … end -- Count from 1 to 10
```

- Tables:

```
local options = { [1] = ‘value’, [‘key’] = 1, -- Numbers starts from 1
  another_key = function(task) … end, -- Functions can be values
  [2] = {} -- Other tables can be values
} -- Can have both numbers and strings as key and anything as values
```

- Functions:

```
local function something(task) -- Normal definition
  local cb = function(data) -- Functions can be nested
  …
  end
end
```

- Closures:

```
local function gen_closure(option)
  local ret = false -- Local variable
  return function(task)
    task:do_something(ret, option) -- Both ‘ret’ and ‘option’ are accessible here
  end
end
rspamd_config.SYMBOL = gen_closure(‘some_option’)
```

Pure LUA functions

Generic recommendations

- Use **local** whenever possible (otherwise, global variables are expensive)
- Callbacks, closures and recursion are generally cheap (when using **LuaJIT**)
- Do not mix string and number keys in tables, that makes them hard to iterate
- **ipairs** and **pairs** are not equal
- Strings are **constant** in LUA

Regexp rules

Types

- Can work with the following elements:
  - Headers: `Message-Id=/^something$/H`
  - Mime parts: `/some word/P`
  - Raw messages: `/some pattern/M`
  - URLs: `/example.com/U`
- Some new flags are added:
  - UTF8 flag: `/u`

Regexp rules

Generic information

- Can be combined using the following operators:
  - **AND**: `/something/P && Subject=/some/H`
  - **OR:** `/something/P || Subject=/some/H`
  - **NOT:** `!/something/P`
  - **PLUS:** `/A/P + /B/P + /C/P >= 2`
- Priority goes as following: NOT ➡️ AND ➡️ OR ➡️ PLUS
- Braces can change priority: `!A AND (B OR C)`

Regexp rules

Performance considerations

- Avoid message regexps at any cost (use trie instead)
- Regexp expressions are highly optimised in rspamd and unnecessary evaluations are not performed
- UTF regexps are more expensive than default ones (but could be useful sometimes)
- **Always** use the appropriate type of expression (e.g. url for links and part for textual content)

Trie matching

- Perfect for fast raw message and text pattern matching
- Scales almost linearly from input size (*aho-corasic* algorithm)
- Can handle thousands and hundreds thousands patterns (is a base for all antivirus scanners)
- Highly optimised for 64 bits systems

Questions?

Vsevolod Stakhov
https://rspamd.com