Profiling real-world golang service

Few words about me

Mikhail Tsimbalov @tmvrus

- 15 years in IT
- sysadmin / developer / CTO
- Go-lover since 2015

Long story short...

- little and proud startup

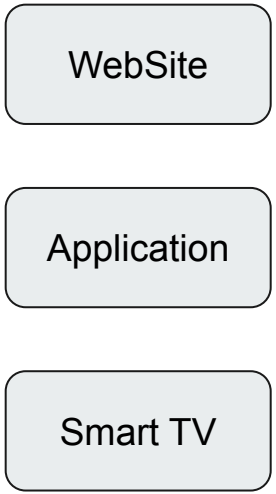
Long story short...

- little and proud startup
- modern stack with ML

Long story short...

- little and proud startup
- modern stack with ML
- got performance issue and solve it

Clients with traffic



Partners with ad

Our traffic exchange

WebSite

Application

Smart TV

Advertiser

Exchange

Ad Network

Client ad request

WebSite

Application

Smart TV

Ad Request

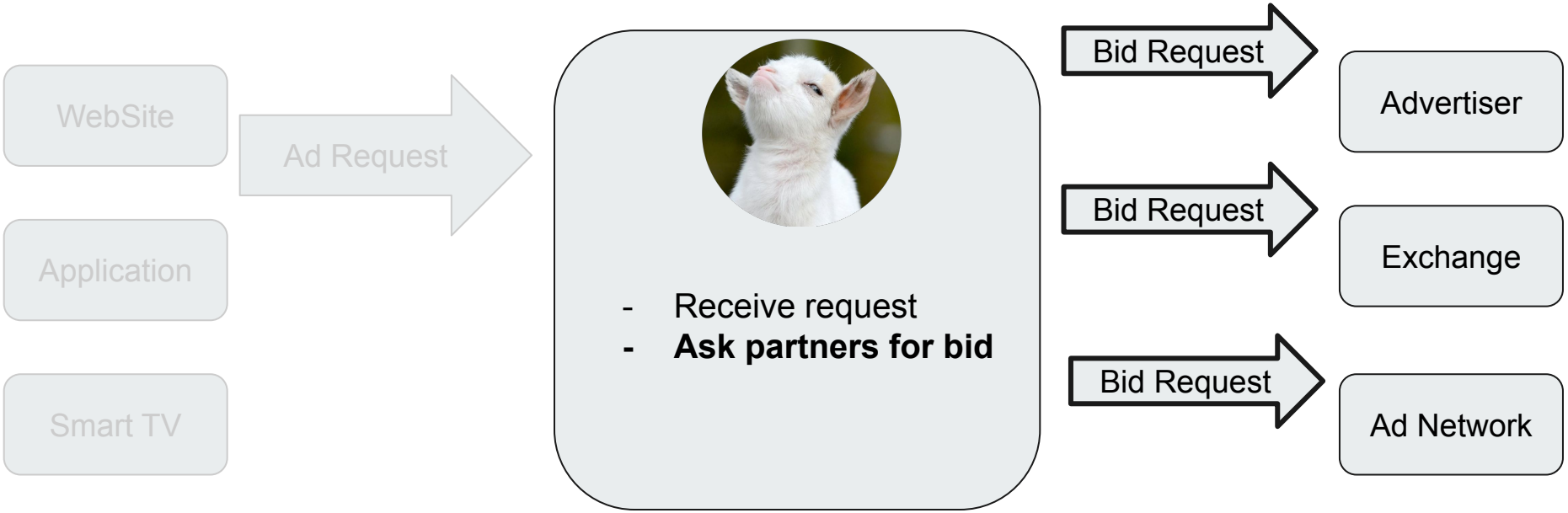
- **Receive ad request**

Advertiser

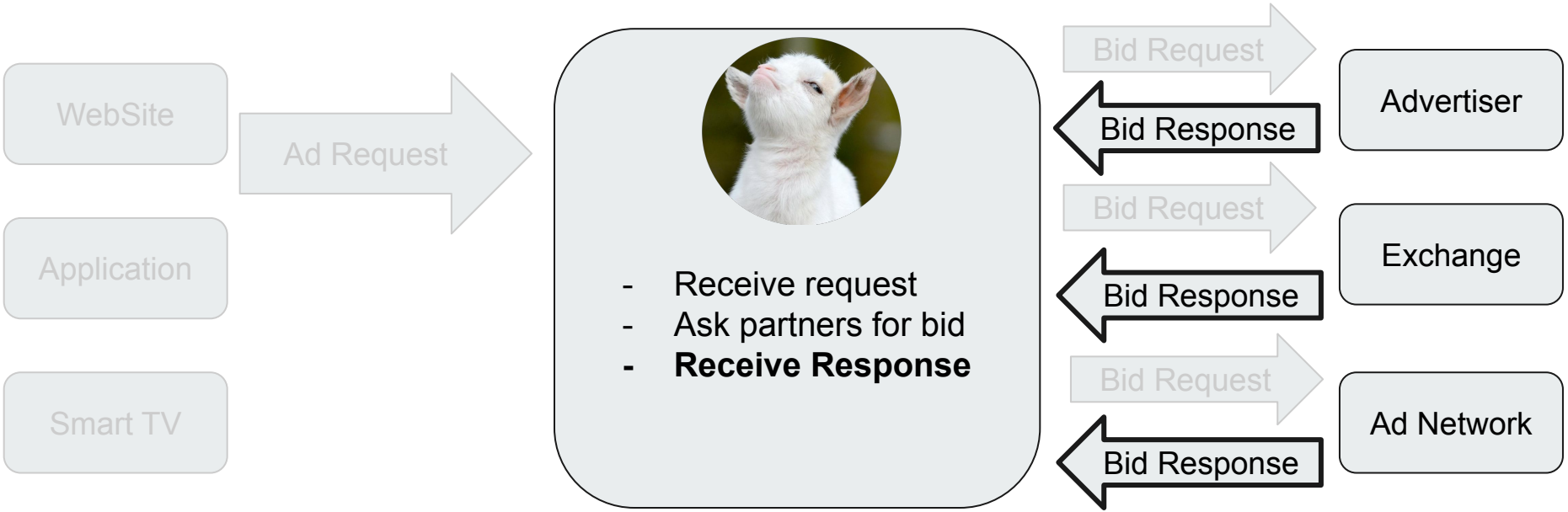
Exchange

Ad Network

Bid request

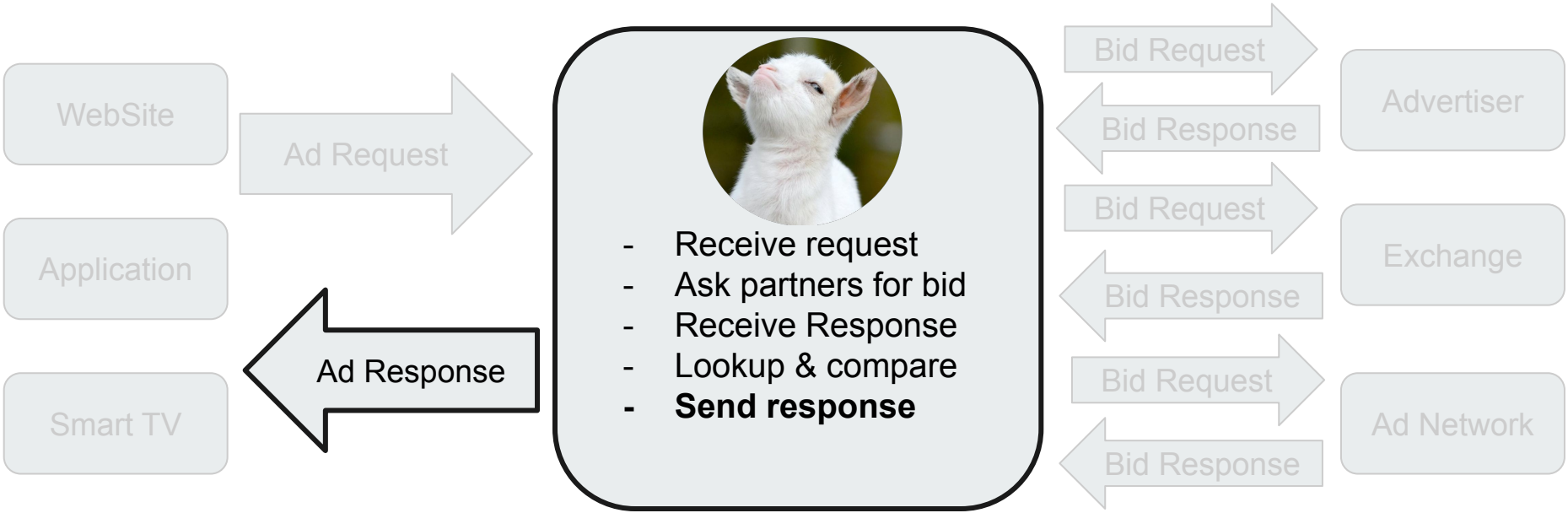

WebSite
Application
Smart TV
Ad Request
- Receive request
- **Ask partners for bid**
Bid Request
Bid Request
Bid Request
Advertiser
Exchange
Ad Network

Bid response

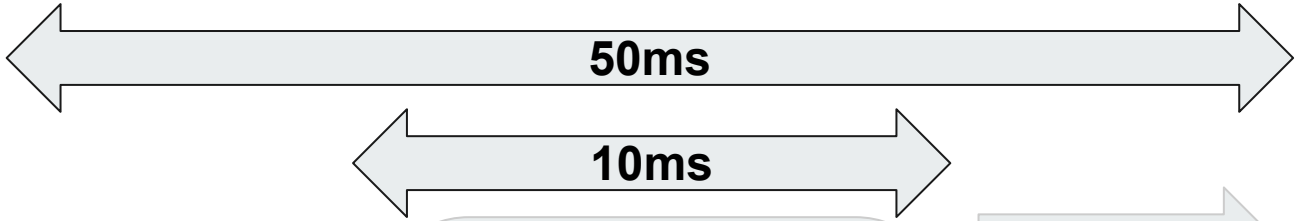


Lookup & compare

Response to client



Time limitation



WebSite

Application

Smart TV

Ad Request

Ad Response

- Receive request
- Ask partners for bid
- Receive Response
- Lookup & compare
- **Send response**

Bid Request

Bid Response

Bid Request

Bid Response

Bid Request

Bid Response

Advertiser

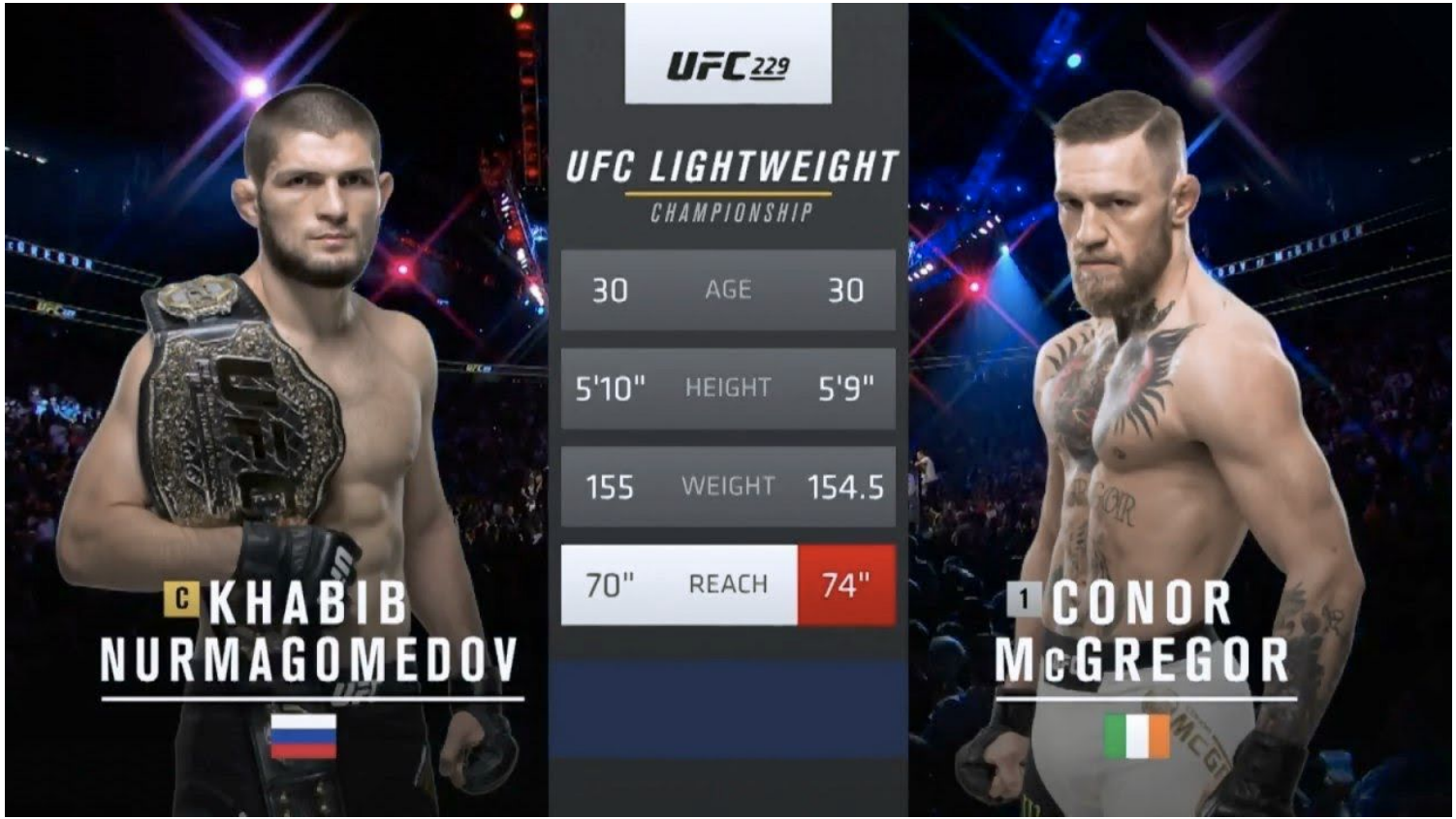
Exchange

Ad Network

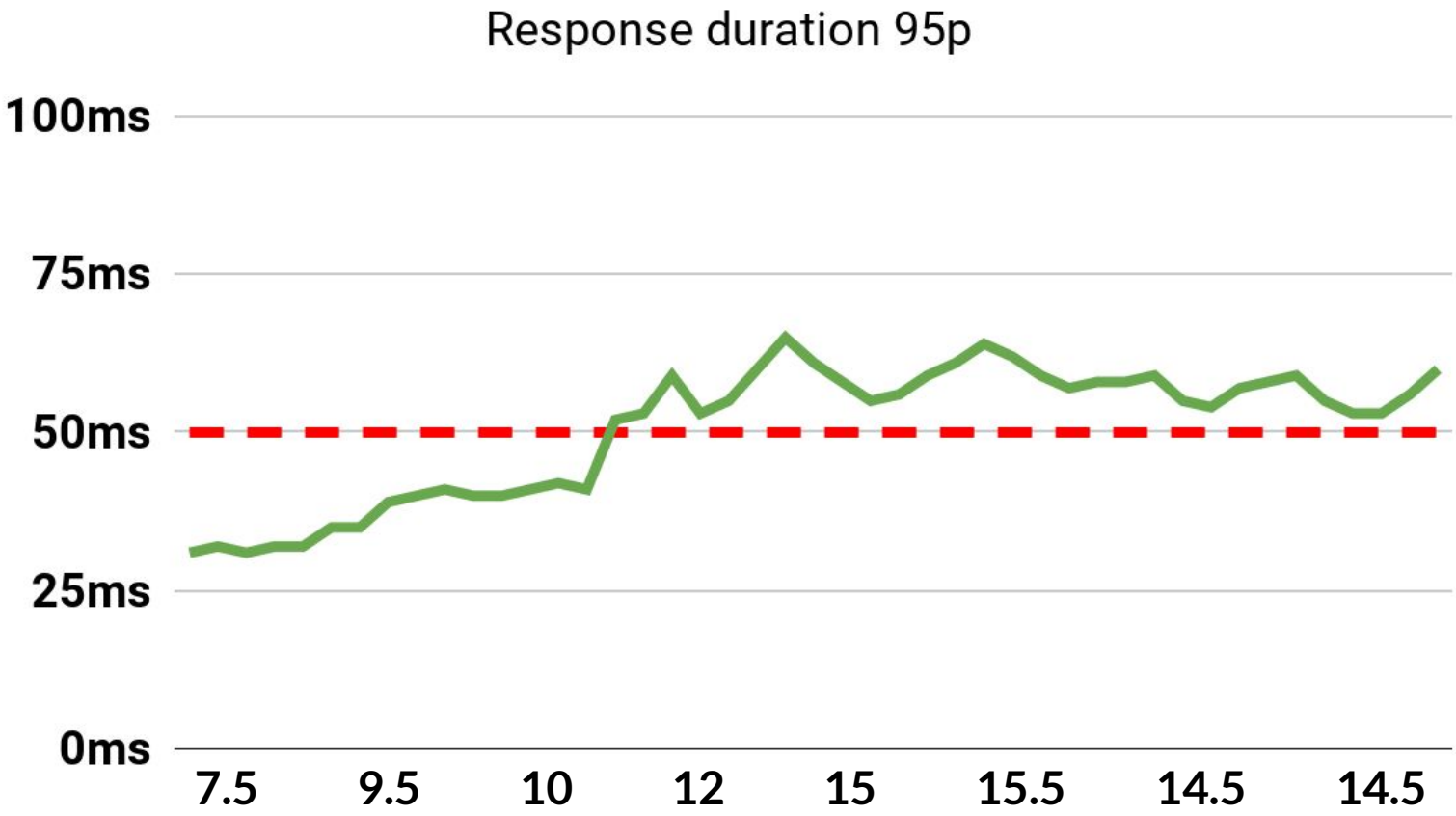
And here it comes big traffic volume #1

And here it comes big traffic volume #2

And here it comes big traffic volume #3



We fallen


Response duration 95p
100ms
75ms
50ms
25ms
0ms
7.5
9.5
10
12
15
15.5
14.5
14.5

We need more money, but

Let’s do profiling

- what is application doing?

Let’s do profiling

- what is application doing?
- **how much it costs?**

Let’s do profiling

- what is application doing?
- how much it costs?
- **can we improve it?**

Let's do profiling

- what is application doing?
- how much it costs?
- can we improve it?

Golang profiler https://git.io/JelHg

Golang profiler

- **SIGPROF signal handler**

Golang profiler

- SIGPROF signal handler
- **collect stack trace 100 times per second, by default**

Golang profiler

- SIGPROF signal handler
- collect stack trace 100 times per second, by default
- **sampling and save collected profile**

How to profile with pprof

Remote application

```
import _ “net/http/pprof”
curl http://some.host/debug/pprof/{PROFILE_TYPE}
```

- profile
- heap
- block, mutex, goroutine ...

How to profile with pprof

Remote application

```
import _ “net/http/pprof”
curl http://some.host/debug/pprof/{PROFILE_TYPE}
```

Local application

```
import “runtime/pprof”
pprof.StartCpuProfile(io.Writer)
```

How to profile with pprof

Remote application

```
import _ “net/http/pprof”
curl http://some.host/debug/pprof/{PROFILE_TYPE}
```

Local application

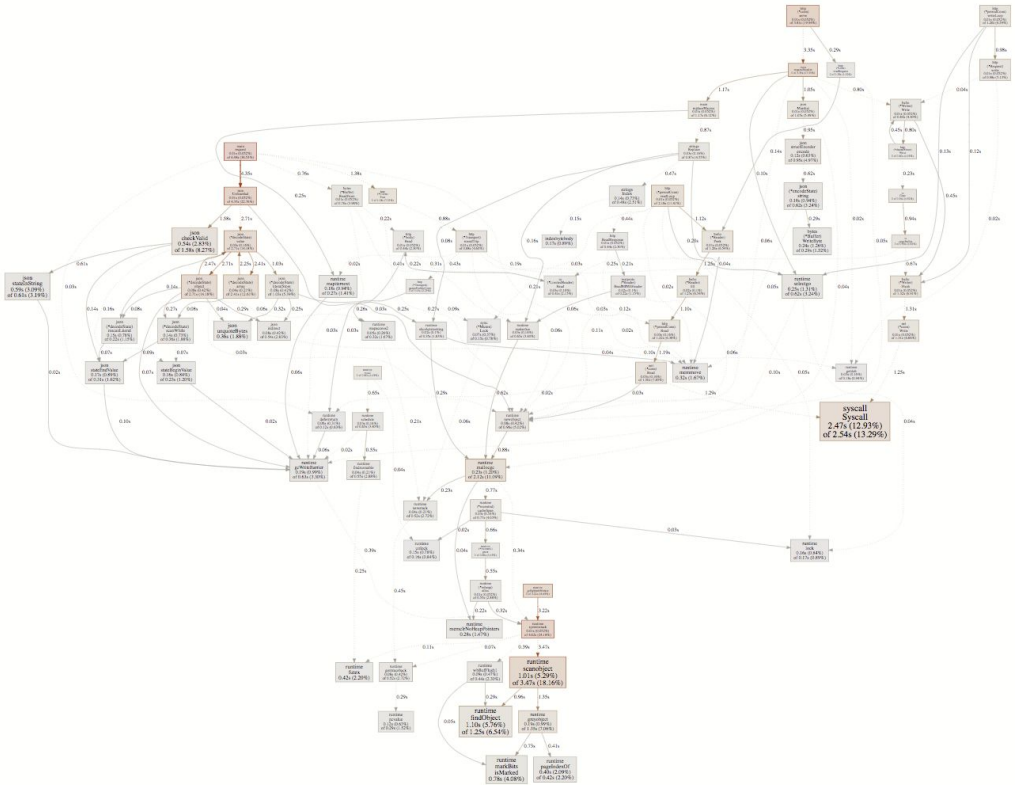
```
import “runtime/pprof”
pprof.StartCpuProfile(output)
```

Piece of code

```
go test -bench=SomeBenchmark -cpuprofile=file.name
```

Function call graph

```
> go tool pprof file.name|URL
> (pprof) web
```



Look at the top

```
> (pprof) top
```

flat	flat%	sum%	cum	cum%	
2740ms	14.40%	14.40%	2890ms	15.19%	syscall.Syscall
840ms	4.41%	18.81%	960ms	5.04%	runtime.findObject
580ms	3.05%	21.86%	600ms	3.15%	encoding/json.stateInString
550ms	2.89%	24.75%	550ms	2.89%	runtime.markBits.isMarked
540ms	2.84%	27.59%	1580ms	8.30%	encoding/json.checkValid
540ms	2.84%	30.43%	540ms	2.84%	runtime.futex
540ms	2.84%	33.26%	1980ms	10.40%	runtime.scanobject
430ms	2.26%	35.52%	430ms	2.26%	runtime.memmove
300ms	1.58%	37.10%	300ms	1.58%	encoding/json.unquoteBytes
290ms	1.52%	38.62%	340ms	1.79%	encoding/json.(*decodeState).rescanLiteral

Look at the top

```
> (pprof) top
```

flat	flat%	sum%	cum	cum%	
2740ms	14.40%	14.40%	**2890ms**	15.19%	syscall.Syscall
840ms	4.41%	18.81%	**960ms**	5.04%	runtime.findObject
580ms	3.05%	21.86%	**600ms**	3.15%	encoding/json.stateInString
550ms	2.89%	24.75%	**550ms**	2.89%	runtime.markBits.isMarked
540ms	2.84%	27.59%	**1580ms**	8.30%	encoding/json.checkValid
540ms	2.84%	30.43%	**540ms**	2.84%	runtime.futex
540ms	2.84%	33.26%	**1980ms**	10.40%	runtime.scanobject
430ms	2.26%	35.52%	**430ms**	2.26%	runtime.memmove
300ms	1.58%	37.10%	**300ms**	1.58%	encoding/json.unquoteBytes
290ms	1.52%	38.62%	**340ms**	1.79%	encoding/json.(*decodeState).rescanLiteral

Look at the top

```
> (pprof) top
```

flat	flat%	sum%	cum	cum%	
2740ms	14.40%	14.40%	2890ms	15.19%	syscall.Syscall
840ms	4.41%	18.81%	960ms	5.04%	runtime.findObject
580ms	**3.05%**	**21.86%**	**600ms**	**3.15%**	**encoding/json.stateInString**
550ms	2.89%	24.75%	550ms	2.89%	runtime.markBits.isMarked
540ms	**2.84%**	**27.59%**	**1580ms**	**8.30%**	**encoding/json.checkValid**
540ms	2.84%	30.43%	540ms	2.84%	runtime.futex
540ms	2.84%	33.26%	1980ms	10.40%	runtime.scanobject
430ms	2.26%	35.52%	430ms	2.26%	runtime.memmove
300ms	**1.58%**	**37.10%**	**300ms**	**1.58%**	**encoding/json.unquoteBytes**
290ms	**1.52%**	**38.62%**	**340ms**	**1.79%**	**encoding/json.(*decodeState).rescanLiteral**

Cumulative values

```
> (pprof) top5 -cum
```

flat	flat%	sum%	cum	cum%	
0	0%	0%	7.17s	37.68%	main.request
0.01s	**0.053%**	**0.053%**	**4.30s**	**22.60%**	**encoding/json.Unmarshal**
0.01s	0.053%	0.11%	4s	21.02%	net/http.(*conn).serve
0.03s	0.16%	0.26%	3.40s	17.87%	runtime.systemstack
0	0%	0.26%	3.30s	17.34%	net/http.(*ServeMux).ServeHTTP

Function listing

```
> (pprof) list main.request
```

```
     4.30s    155:  if err := json.Unmarshal(data, &bid); err != nil {
         .    156:      err = Wrap(err, “failed to Unmarshal partner response”)
         .    157:      return err
         .    158:  }
```

Can we do Unmarshal better/faster?

- code our custom solution
- get third-party package

github.com/json-iterator/go

- drop-in replace

github.com/json-iterator/go

- drop-in replace

- **no code generation like easyjson or ffjson**

github.com/json-iterator/go

- drop-in replace
- no code generation like easyjson or ffjson
- **it 4-times faster** https://git.io/JelN8

```
BenchmarkEncodingJSONUnmarshal        580          2212113 ns/op
BenchmarkJsoniterUnmarshal           2005           629120 ns/op
```

Cumulative values json-iter

```
> (pprof) top5 -cum
```

flat	flat%	sum%	cum	cum%	
0.04s	**0.25%**	**0.25%**	**4.25s**	**26.25%**	**main.request**
0	0%	0.25%	3.97s	24.52%	runtime.systemstack
0	0%	0.25%	3.57s	22.05%	net/http.(*conn).serve
2.91s	17.97%	18.22%	3.03s	18.72%	syscall.Syscall
0.57s	3.52%	21.74%	2.74s	16.92%	runtime.scanobject

Function listing

```
> (pprof) list main.request
```

```
     1.24s    155:  if err := jsoniter.Unmarshal(data, &bid); err != nil {
         .    156:      err = Wrap(err, “failed to Unmarshal partner response”)
         .    157:      return err
         .    158:  }
```

Why is json-iter faster?

- linear algorithm

Why is json-iter faster?

- linear algorithm
- **no separate validation phase**

Why is json-iter faster?

- linear algorithm
- no separate validation phase
- **sync.Pool**

Look at the top with json-iter

```
> (pprof) top
```

flat	flat%	sum%	cum	cum%	
2910ms	17.97%	17.97%	3030ms	18.72%	syscall.Syscall
1010ms	6.24%	24.21%	1190ms	7.35%	runtime.findObject
670ms	4.14%	28.35%	670ms	4.14%	runtime.futex
670ms	4.14%	32.49%	670ms	4.14%	runtime.markBits.isMarked
570ms	3.52%	36.01%	2740ms	16.92%	runtime.scanobject
310ms	1.91%	37.92%	1980ms	12.23%	runtime.mallocgc
290ms	1.79%	39.72%	300ms	1.85%	runtime.pageIndexOf
280ms	1.73%	41.45%	280ms	1.73%	runtime.memmove
250ms	1.54%	42.99%	250ms	1.54%	runtime.nextFreeFast
240ms	1.48%	44.47%	240ms	1.48%	runtime.memclrNoHeapPointers

Look at the top with json-iter

```
> (pprof) top
```

flat	flat%	sum%	cum	cum%	
2910ms	17.97%	17.97%	3030ms	18.72%	syscall.Syscall
1010ms	6.24%	24.21%	1190ms	7.35%	runtime.findObject
670ms	4.14%	28.35%	670ms	4.14%	runtime.futex
670ms	4.14%	32.49%	670ms	4.14%	runtime.markBits.isMarked
570ms	3.52%	36.01%	2740ms	16.92%	runtime.scanobject
310ms	1.91%	37.92%	1980ms	12.23%	runtime.mallocgc
290ms	1.79%	39.72%	300ms	1.85%	runtime.pageIndexOf
280ms	1.73%	41.45%	280ms	1.73%	runtime.memmove
250ms	1.54%	42.99%	250ms	1.54%	runtime.nextFreeFast
240ms	1.48%	44.47%	240ms	1.48%	runtime.memclrNoHeapPointers

Heap profiling

```
http://some.host/debug/pprof/profile
```

Heap profiling

~~`http://some.host/debug/pprof/profile`~~

`http://some.host/debug/pprof/heap`

Heap profiling

```
> go tool pprof http://some.host/debug/pprof/heap
```

```
> (pprof) top5
```

flat	flat%	sum%	cum	cum%	
442MB	97.91%	97.91%	442MB	97.91%	main.loadBidSet
7.53MB	2.29%	95.42%	7.53MB	2.29%	bufio.NewWriterSize
6.53MB	1.99%	97.40%	6.53MB	1.99%	bufio.NewReaderSize
2.50MB	0.76%	98.16%	15.04MB	4.58%	net/http.(*Transport).dialConn
0	0%	98.16%	306MB	93.13%	main.init.0

BidSet

```
map[string]bidPayload


type bidPayload struct {
  PartnerID, Currency, Custom  string
  BidId, SourceID int
  AdMarkup   []byte
  Tags       []string
}

key := fmt.Sprintf(“%d%d”, BidID, SourceID)
```

BidSet

```
map[uint64]bidPayload


type bidPayload struct {
  PartnerID, Currency, Custom  string
  BidId, SourceID uint32
  AdMarkup   []byte
  Tags       []string
}

key := (uint64(BidID) << 32) | uint64(SourceID)
```

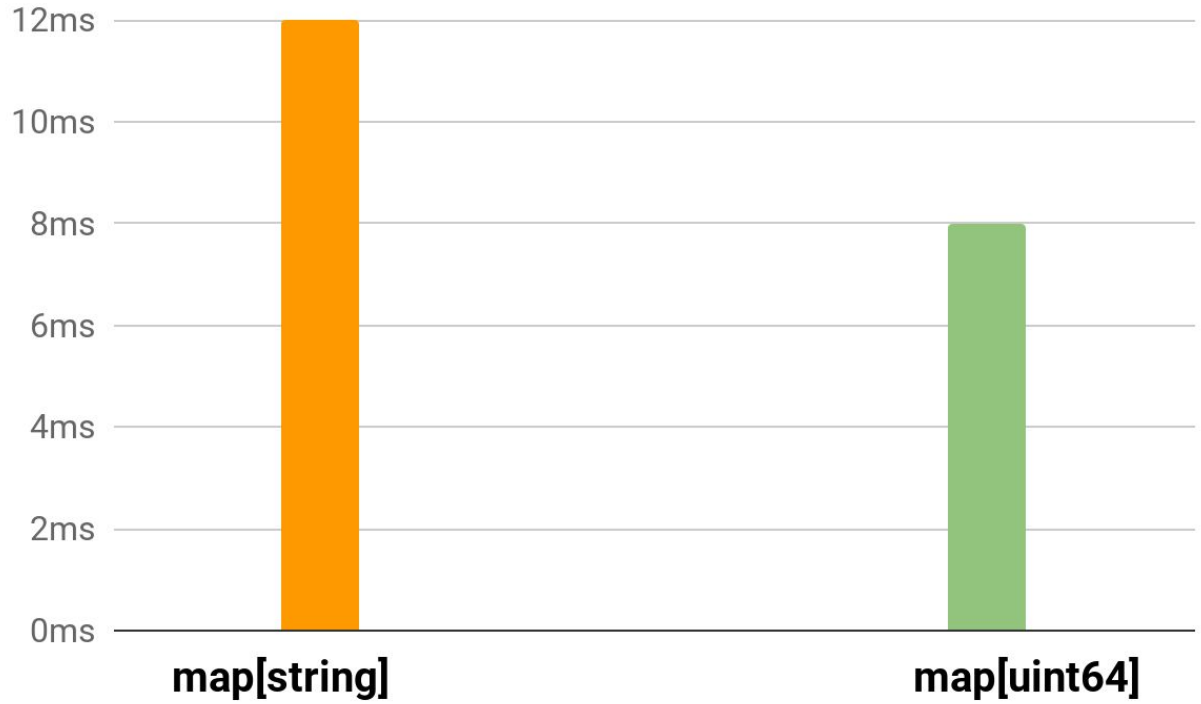
Profit #1

reduce memory usage

```
      flat  flat%   sum%        cum   cum%
   306MB 93.13% 93.13%      306MB 93.13%  main.loadBidSet
 7.53MB  2.29% 95.42%     7.53MB  2.29%  bufio.NewWriterSize
 6.53MB  1.99% 97.40%     6.53MB  1.99%  bufio.NewReaderSize
 2.50MB  0.76% 98.16%    15.04MB  4.58%  net/http.(*Transport).dialConn
      0     0% 98.16%      306MB 93.13%  main.init.0
```

Profit #2

`reduce gc duration`



Profit #3

reduce lookup time

BenchmarkMapUint	**11430424**	**103 ns/op**
BenchmarkMapString	5238861	223 ns/op

https://git.io/Je8m8

Runtime map lookup optimization

- map[**uint32**]

  mapaccess1_**fast32**(t *maptype, h *hmap, **key uint32**) unsafe.Pointer

  mapaccess2_**fast32**(t *maptype, h *hmap, **key uint32**) (unsafe.Pointer, bool)

Runtime map lookup optimization

- map[**uint32**]
  - mapaccess1_**fast32**(t *maptype, h *hmap, **key uint32**) unsafe.Pointer
  - mapaccess2_**fast32**(t *maptype, h *hmap, **key uint32**) (unsafe.Pointer, bool)

- map[**uint64**]
  - mapaccess1_**fast64**(t *maptype, h *hmap, **key uint64**) unsafe.Pointer
  - mapaccess1_**fast64**(t *maptype, h *hmap, **key uint64**) (unsafe.Pointer, bool)

Runtime map lookup optimization

- map[**uint32**]
  mapaccess1_**fast32**(t *maptype, h *hmap, **key uint32**) unsafe.Pointer
  mapaccess2_**fast32**(t *maptype, h *hmap, **key uint32**) (unsafe.Pointer, bool)

- map[**uint64**]
  mapaccess1_**fast64**(t *maptype, h *hmap, **key uint64**) unsafe.Pointer
  mapaccess1_**fast64**(t *maptype, h *hmap, **key uint64**) (unsafe.Pointer, bool)

- map[**string**]
  mapaccess1_**faststr**(t *maptype, h *hmap, **key string**) unsafe.Pointer
  mapaccess2_**faststr**(t *maptype, h *hmap, **key string**) (unsafe.Pointer, bool)

Runtime map lookup optimization

- map[**uint32**]
  mapaccess1_**fast32**(t *maptype, h *hmap, **key uint32**) unsafe.Pointer
  mapaccess2_**fast32**(t *maptype, h *hmap, **key uint32**) (unsafe.Pointer, bool)

- map[**uint64**]
  mapaccess1_**fast64**(t *maptype, h *hmap, **key uint64**) unsafe.Pointer
  mapaccess1_**fast64**(t *maptype, h *hmap, **key uint64**) (unsafe.Pointer, bool)

- map[**string**]
  mapaccess1_**faststr**(t *maptype, h *hmap, **key string**) unsafe.Pointer
  mapaccess2_**faststr**(t *maptype, h *hmap, **key string**) (unsafe.Pointer, bool)

- map[**float, struct...**]
  mapaccess1(t *maptype, h *hmap, **key unsafe.Pointer**) unsafe.Pointer
  mapaccess2(t *maptype, h *hmap, **key unsafe.Pointer**) (unsafe.Pointer, bool)

Look at heap again

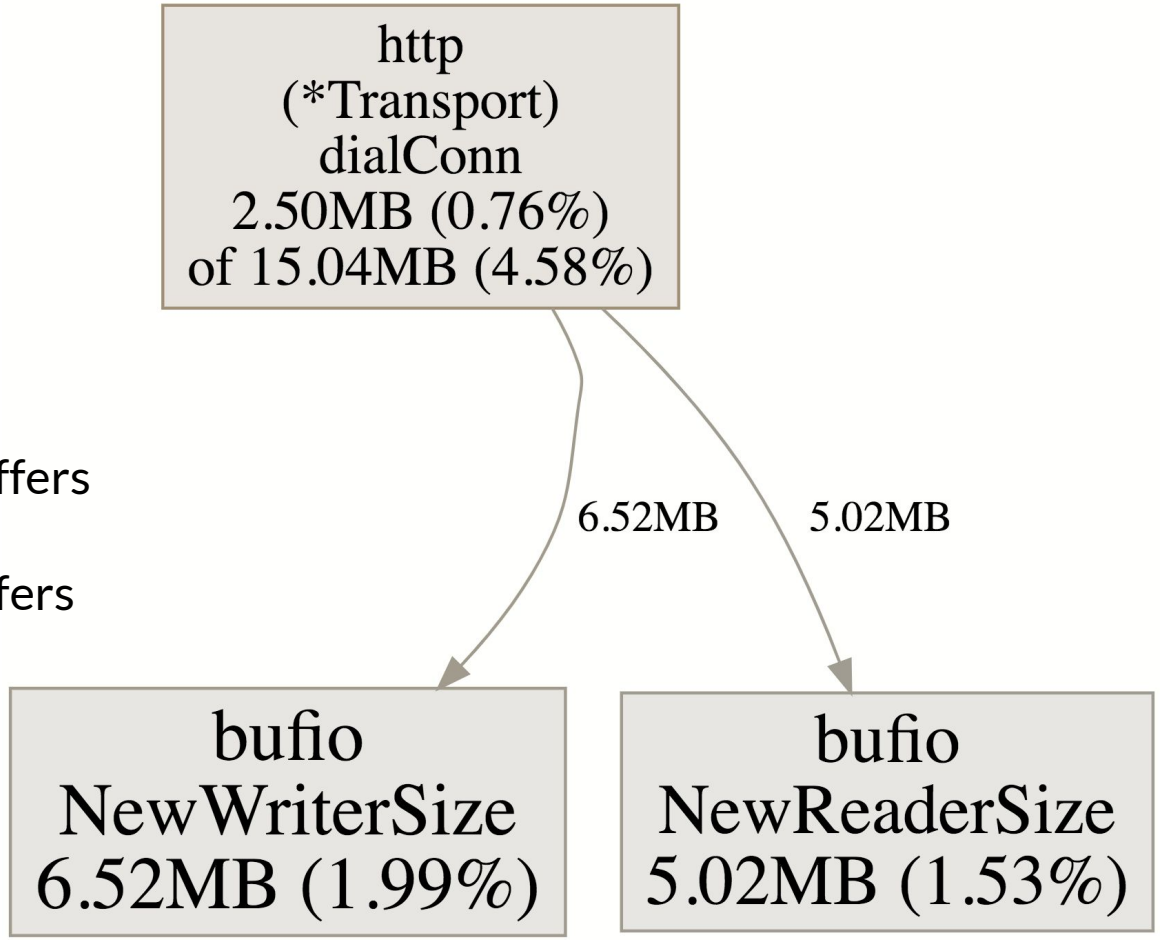
```
(pprof) top5 -cum
```

flat	flat%	sum%	cum	cum%	
0	0%	0%	306MB	93.13%	main.init.0
306MB	93.13%	93.13%	306MB	93.13%	main.loadBidSet
0	0%	93.13%	306MB	93.13%	runtime.doInit
0	0%	93.13%	306MB	93.13%	runtime.main
2.50MB	**0.76%**	**93.89%**	**15.04MB**	**4.58%**	**net/http.(*Transport).dialConn**

dialConn call graph

`(pprof) web dialConn`

- 0.76% - 2.5MB inplace
- 1.53% - 6.52MB for write buffers
- 1.99% - 5.02MB for read buffers



Zero allocations function

- **reuse memory via sync.Pool**
  buffers, structures...

Zero allocations function

- reuse memory via sync.Pool
  buffers, structures...

- **reduce GC-pressure**
  less allocations, less work

Zero allocations function

- reuse memory via sync.Pool
  buffers, structures...

- reduce GC-pressure
  less allocations, less work

- **fastest execution**
  no context switch

Zero allocations http client

github.com/valyala/fasthttp

BenchmarkFastHTTP	**344619**	**3396**	**ns/op**	**122**	**B/op**	**6**	**allocs/op**
BenchmarkNetHTTP	112113	9179	ns/op	16820	B/op	43	allocs/op

https://git.io/Je8nD

https://youtu.be/fg3JPUswiek

Little example

- **net/http request**

```
http.Post(url, "application-json",requestBody)
```

Little example

- net/http request

```
http.Post(url, "application-json",requestBody)
```

- fasthttp request

```
req := fasthttp.AcquireRequest()
resp := fasthttp.AcquireResponse()
```

Little example

- net/http request

```
http.Post(url, "application-json",requestBody)
```

- **fasthttp request**

```
req := fasthttp.AcquireRequest()
resp := fasthttp.AcquireResponse()

fasthttp.Do(req, resp)
```

Little example

- net/http request

```
http.Post(url, "application-json",requestBody)
```

- **fasthttp request**

```
req := fasthttp.AcquireRequest()
resp := fasthttp.AcquireResponse()

fasthttp.Do(req, resp)

fasthttp.ReleaseRequest(req)
fasthttp.ReleaseResponse(resp)
```

Look at heap again

```
(pprof) top6 -cum
```

flat	flat%	sum%	cum	cum%	
0	0%	0%	306.50MB	95.31%	runtime.main
0	0%	0%	306MB	95.15%	main.init.0
306MB	95.15%	95.15%	306MB	95.15%	main.loadBidSet
0	0%	95.15%	306MB	95.15%	runtime.doInit
0	0%	95.15%	11.57MB	3.60%	main.request
6.05MB	**1.88%**	**97.03%**	**6.05MB**	**1.88%**	**valyala/bytebufferpool.(*Pool).Get**

Bytebufferpool

- backed on sync.Pool

Bytebufferpool

- backed on sync.Pool
- **collects buffer size statistics**

Bytebufferpool

- backed on sync.Pool
- collects buffer size statistics
- **calculates optimal default buffer size**

Bytebufferpool

- backed on sync.Pool
- collects buffer size statistics
- calculates optimal default buffer size
- **avoid buffer appending**

Look at the top

```
> (pprof) top
```

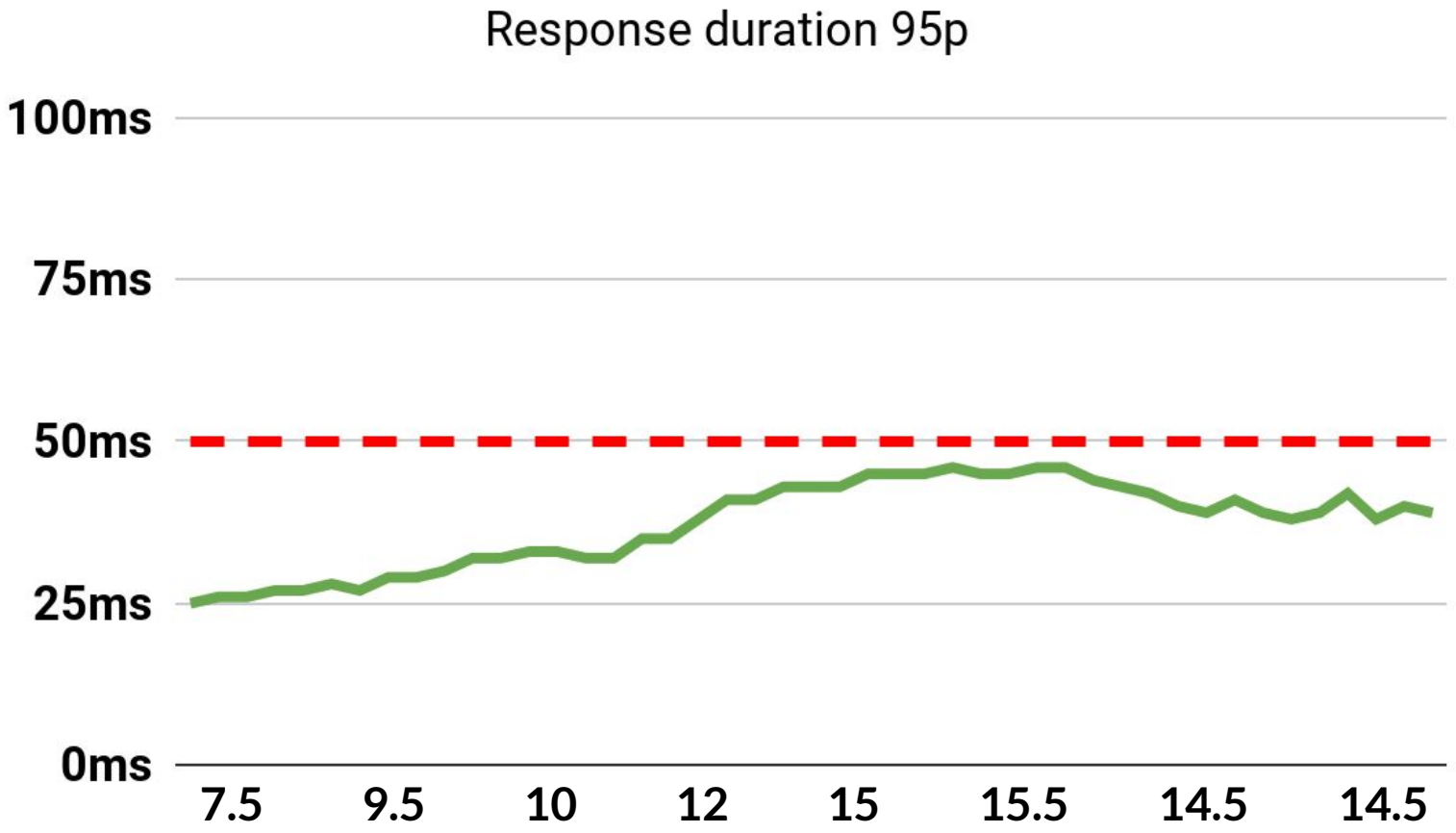
flat	flat%	sum%	cum	cum%	
3.54s	22.87%	22.87%	3.68s	23.77%	syscall.Syscall
1.07s	6.91%	29.78%	1.07s	6.91%	runtime.futex
0.58s	**3.75%**	**33.53%**	**0.58s**	**3.75%**	**runtime.memmove**
0.50s	3.23%	36.76%	0.96s	6.20%	json-iter.(*Iterator).ReadString
0.38s	2.45%	39.21%	0.38s	2.45%	json-iter.(*Iterator).nextToken
0.37s	**2.39%**	**41.60%**	**0.37s**	**2.39%**	**runtime.nextFreeFast**
0.30s	**1.94%**	**43.54%**	**1.67s**	**10.79%**	**runtime.mallocgc**
0.30s	1.94%	45.48%	0.74s	4.78%	strings.Index
0.27s	1.74%	47.22%	0.27s	1.74%	indexbytebody
0.21s	1.36%	50.19%	0.24s	1.55%	runtime.mapiternext

Look at the top

```
> (pprof) top
```

flat	flat%	sum%	cum	cum%	
3.54s	22.87%	22.87%	3.68s	23.77%	syscall.Syscall
1.07s	6.91%	29.78%	1.07s	6.91%	runtime.futex
0.58s	3.75%	33.53%	0.58s	3.75%	runtime.memmove
0.50s	**3.23%**	**36.76%**	**0.96s**	**6.20%**	**json-iter.(*Iterator).ReadString**
0.38s	**2.45%**	**39.21%**	**0.38s**	**2.45%**	**json-iter.(*Iterator).nextToken**
0.37s	2.39%	41.60%	0.37s	2.39%	runtime.nextFreeFast
0.30s	1.94%	43.54%	1.67s	10.79%	runtime.mallocgc
0.30s	**1.94%**	**45.48%**	**0.74s**	**4.78%**	**strings.Index**
0.27s	**1.74%**	**47.22%**	**0.27s**	**1.74%**	**indexbytebody**
0.21s	**1.36%**	**50.19%**	**0.24s**	**1.55%**	**runtime.mapiternext**

Result


Response duration 95p
100ms
75ms
50ms
25ms
0ms
7.5
9.5
10
12
15
15.5
14.5
14.5

Summary

- Premature optimization is the root of all evil
- Profiling is easy with go
- Benchmarking easy to
- Profiling and benchmarking amazing together
- Third-party packages are often useful
- Allocations isn't always a bad thing
- But it is better if there are fewer

Questions

https://git.io/JeBiI