High Performance Systems in Go

Derek Collison
April 24, 2014
GopherCon

About

Derek Collison

- Architected/Built TIBCO Rendezvous and EMS Messaging Systems
- Designed and Built CloudFoundry at VMware
- Co-founded AJAX APIs group at Google
- Distributed Systems
- Founder of Apcera, Inc. in San Francisco, CA
- @derekcollison
- derek@apcera.com

Why Go?

- Simple Compiled Language
- Good Standard Library
- Concurrency
- Synchronous Programming Model
- Garbage Collection
- STACKS!

Why Go?

- Not C/C++
- Not Java (or any JVM based language)
- Not Ruby/Python/Node.js

Derek Collison
@derekcollison

Prediction: Go will become the dominant language for systems work in IaaS, Orchestration, and PaaS in 24 months. #golang

Reply · Delete · Favorite · More

63 RETWEETS | **34** FAVORITES

9:00 AM - 11 Sep 12

Related headlines

The Go language is gaining momentum among PaaS and I…
Gigaom @gigaom

What about
High Performance?

NATS

NATS

NATS is an open-source, lightweight, publish-subscribe & distributed queueing messaging system.

Get the latest stable version.

DOWNLOAD
NATS v0.5

More server and client options.

Downloads »

Learn more about **NATS** features.

Documentation »

SUPPORTED BY

apcera

NATS Messaging 101

- Subject-Based
- Publish-Subscribe
- Distributing Queueing
- TCP/IP Overlay
- Clustered Servers
- Multiple Clients (Go, Node.js, Java, Ruby)

NATS

NATS for Go

Basic Encoded Usage

```go
nc, _ := nats.Connect(nats.DefaultURL)
c, _ := nats.NewEncodedConn(nc, "json")
defer c.Close()

// Simple Publisher
c.Publish("foo", "Hello World")

// Simple Async Subscriber
c.Subscribe("foo", func(s string) {
    fmt.Printf("Received a message: %s\n", s)
})

// EncodedConn can Publish any raw Go type using the registered Encoder
type person struct {
     Name     string
     Address  string
     Age      int
}

// Go type Subscriber
c.Subscribe("hello", func(p *person) {
    fmt.Printf("Received a person: %+v\n", p)
})

me := &person{Name: "derek", Age: 22, Address: "585 Howard Street, San Francisco, CA"}

// Go type Publisher
c.Publish("hello", me)
```

Using Go Channels (netchan)

```
nc, _ := nats.Connect(nats.DefaultURL)
c, _ := nats.NewEncodedConn(nc, "json")
defer c.Close()

type person struct {
     Name     string
     Address  string
     Age      int
}

recvCh := make(chan *person)
c.BindRecvChan("hello", recvCh)

sendCh := make(chan *person)
c.BindSendChan("hello", sendCh)

me := &person{Name: "derek", Age: 22, Address: "585 Howard Street"}

// Send via Go channels
sendCh <- me

// Receive via Go channels
who := <- recvCh
```

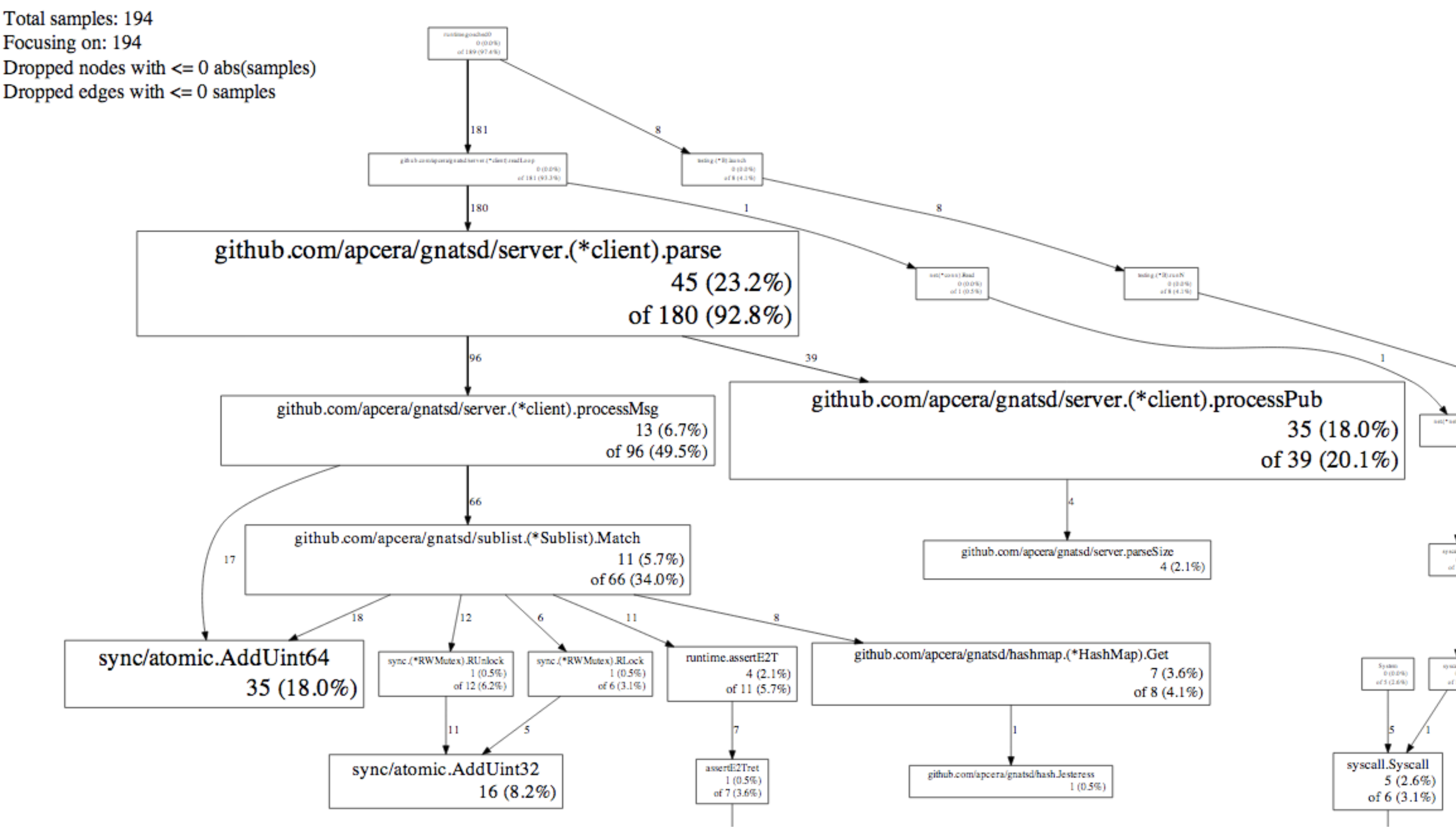
NATS

- Originally written to support CloudFoundry
- In use by CloudFoundry, Baidu, Apcera and others
- Written first in Ruby -> 150k msgs/sec
- Rewritten at Apcera in Go (Client and Server)
- First pass -> 500k msgs/sec
- **Current Performance** -> **5-6m msgs/sec**

Tuning NATS (gnatsd)

or

how to get from 500k to 6m


Total samples: 194
Focusing on: 194
Dropped nodes with <= 0 abs(samples)
Dropped edges with <= 0 samples
181
8
180
1
8
github.com/apcera/gnatsd/server.(*client).parse
45 (23.2%)
of 180 (92.8%)
96
39
1
github.com/apcera/gnatsd/server.(*client).processMsg
13 (6.7%)
of 96 (49.5%)
github.com/apcera/gnatsd/server.(*client).processPub
35 (18.0%)
of 39 (20.1%)
66
4
17
github.com/apcera/gnatsd/sublist.(*Sublist).Match
11 (5.7%)
of 66 (34.0%)
github.com/apcera/gnatsd/server.parseSize
4 (2.1%)
18
12
6
11
8
sync/atomic.AddUint64
35 (18.0%)
sync.(*RWMutex).RUnlock
1 (0.5%)
of 12 (6.2%)
sync.(*RWMutex).RLock
1 (0.5%)
of 6 (3.1%)
runtime.assertE2T
4 (2.1%)
of 11 (5.7%)
github.com/apcera/gnatsd/hashmap.(*HashMap).Get
7 (3.6%)
of 8 (4.1%)
11
5
7
1
5
1
sync/atomic.AddUint32
16 (8.2%)
assertE2Tret
1 (0.5%)
of 7 (3.6%)
github.com/apcera/gnatsd/hash.Jesteress
1 (0.5%)
syscall.Syscall
5 (2.6%)
of 6 (3.1%)

Target Areas

- Shuffling Data
- Protocol Parsing
- Subject/Routing

Target Areas

- Shuffling Data
- **Protocol Parsing**
- Subject/Routing

Protocol Parsing

- NATS is a text based protocol
  - `PUB foo.bar 2\r\nok\r\n`
  - `SUB foo.> 2\r\n`
- Ruby version based on RegEx
- First Go version was port of RegEx
- Current is **zero allocation** byte parser

```
		case OP_P:
			switch b {
			case 'U', 'u':
				c.state = OP_PU
			case 'I', 'i':
				c.state = OP_PI
			case 'O', 'o':
				c.state = OP_PO
			default:
				goto parseErr
			}
		case OP_PU:
			switch b {
			case 'B', 'b':
				c.state = OP_PUB
			default:
				goto parseErr
			}
		case OP_PUB:
			switch b {
			case ' ', '\t':
				c.state = OP_PUB_SPC
			default:
				goto parseErr
			}
		case OP_PUB_SPC:
			switch b {
			case ' ', '\t':
				continue
			default:
				c.state = PUB_ARG
				c.as = i
			}
```

```
		case PUB_ARG:
			switch b {
			case '\r':
				c.drop = 1
			case '\n':
				var arg []byte
				if c.argBuf != nil {
					arg = c.argBuf
				} else {
					arg = buf[c.as : i-c.drop]
				}
				if err := c.processPub(arg); err != nil {
					return err
				}
				c.drop, c.as, c.state = 0, i+1, MSG_PAYLOAD
			default:
				if c.argBuf != nil {
					c.argBuf = append(c.argBuf, b)
				}
			}
		case MSG_PAYLOAD:
			if c.msgBuf != nil {
				c.msgBuf = append(c.msgBuf, b)
				if len(c.msgBuf) >= c.pa.size {
					c.state = MSG_END
				}
			} else if i-c.as >= c.pa.size {
				c.state = MSG_END
			}
		case MSG_END:
```

```go
type pubArg struct {
	subject []byte
	reply   []byte
	sid     []byte
	szb     []byte
	size    int
}

type parseState struct {
	state   int
	as      int
	drop    int
	pa      pubArg
	argBuf  []byte
	msgBuf  []byte
	scratch [MAX_CONTROL_LINE_SIZE]byte
}
```

```go
		// Check for split buffer scenarios for any ARG state.
		if (c.state == SUB_ARG || c.state == UNSUB_ARG || c.state == PUB_ARG ||
			c.state == MSG_ARG || c.state == MINUS_ERR_ARG) && c.argBuf == nil {
			c.argBuf = c.scratch[:0]
			c.argBuf = append(c.argBuf, buf[c.as:(i+1)-c.drop]...)
			// FIXME, check max len
		}
```

Some Tidbits

- Early on, **defer** was costly
- Text based proto needs conversion from ascii to int
    - This was also slow due to allocations in **strconv.ParseInt**

defer

gistfile1.go Go

```go
func deferUnlock(mu sync.Mutex) {
	mu.Lock()
	defer mu.Unlock()
}

func BenchmarkDeferMutex(b *testing.B) {
	var mu sync.Mutex
	b.SetBytes(1)
	for i := 0; i < b.N; i++ {
		deferUnlock(mu)
	}
}

func noDeferUnlock(mu sync.Mutex) {
	mu.Lock()
	mu.Unlock()
}

func BenchmarkNoDeferMutex(b *testing.B) {
	var mu sync.Mutex
	b.SetBytes(1)
	for i := 0; i < b.N; i++ {
		noDeferUnlock(mu)
	}
}
```

defer Results

gistfile1.txt

```
MacbookAir 11" i7 1.7Ghz Haswell
Linux mint15 3.8.0-19

=================
Go version go1.0.3
=================

BenchmarkDeferMutex     10000000               243.0 ns/op        4.11 mops/s
BenchmarkNoDeferMutex   20000000                79.9 ns/op       12.52 mops/s

=================
Go version go1.1.2
=================

BenchmarkDeferMutex     10000000               174.0 ns/op        5.72 mops/s
BenchmarkNoDeferMutex   50000000                65.3 ns/op       15.31 mops/s

=================
Go version go1.2.1
=================

BenchmarkDeferMutex     20000000               137.0 ns/op        7.27 mops/s
BenchmarkNoDeferMutex   50000000                62.8 ns/op       15.92 mops/s
```

golang1.3 looks promising

parseSize

gistfile1.go Go

```go
// Ascii numbers 0-9
const (
	ascii_0 = 48
	ascii_9 = 57
)

// parseSize expects decimal positive numbers. We
// return -1 to signal error
func parseSize(d []byte) (n int) {
	if len(d) == 0 {
		return -1
	}
	for _, dec := range d {
		if dec < ascii_0 || dec > ascii_9 {
			return -1
		}
		n = n*10 + (int(dec) - ascii_0)
	}
	return n
}
```

parseSize
vs
strconv.ParseInt

gistfile1.txt

```
2013 MacbookAir 11" i7 1.7Ghz Haswell
Linux mint15 3.8.0-19

=================
Go version go1.0.3
=================

BenchmarkParseInt       50000000                48.3 ns/op       20.72 mops/s
BenchmarkParseSize      100000000               19.9 ns/op       50.35 mops/s

=================
Go version go1.1.2
=================

BenchmarkParseInt       50000000                36.9 ns/op       27.13 mops/s
BenchmarkParseSize      100000000               10.0 ns/op       99.55 mops/s

=================
Go version go1.2.1
=================

BenchmarkParseInt       50000000                35.0 ns/op       28.61 mops/s
BenchmarkParseSize      100000000               10.0 ns/op       99.52 mops/s
```

Target Areas

- Shuffling Data
- Protocol Parsing
- **Subject/Routing**

Subject Router

- Matches subjects to subscribers
- Utilizes a trie of nodes and hashmaps
- Has a frontend dynamic eviction cache
- Uses []byte as keys (Go’s builtin does not)

Subject Router

- Tried to avoid []byte -> string conversions
- Go's builtin hashmap was slow pre 1.0
- Built using hashing algorithms on []byte
- Built on hashmaps with []byte keys

Hashing Algorithms

gistfile1.txt

```
2013 MacbookAir 11" i7 1.7Ghz Haswell

=================
OSX - Mavericks 10.9.2
Go version go1.2.1
=================

Benchmark_Bernstein_SmallKey    500000000               5.13 ns/op      195.10 mops/s
Benchmark_Murmur3___SmallKey    200000000               8.11 ns/op      123.26 mops/s
Benchmark_FNV1A_____SmallKey    500000000               5.07 ns/op      197.36 mops/s
Benchmark_Meiyan____SmallKey    500000000               4.24 ns/op      236.02 mops/s
Benchmark_Jesteress_SmallKey    500000000               5.32 ns/op      188.08 mops/s
Benchmark_Yorikke___SmallKey    500000000               5.52 ns/op      181.20 mops/s
Benchmark_Bernstein___MedKey    50000000               34.90 ns/op       28.65 mops/s
Benchmark_Murmur3_____MedKey    100000000              17.90 ns/op       55.94 mops/s
Benchmark_FNV1A_______MedKey    50000000               31.90 ns/op       31.37 mops/s
Benchmark_Meiyan______MedKey    200000000               9.28 ns/op      107.76 mops/s
Benchmark_Jesteress___MedKey    200000000               8.15 ns/op      122.65 mops/s
Benchmark_Yorikke_____MedKey    200000000               9.19 ns/op      108.83 mops/s
```

Hashing Algorithms

gistfile1.txt

```
2013 MacbookAir 11" i7 1.7Ghz Haswell

=================
OSX - Mavericks 10.9.2
Go version go1.2.1
=================

Benchmark_Bernstein_SmallKey    500000000               5.13 ns/op      195.10 mops/s
Benchmark_Murmur3___SmallKey    200000000               8.11 ns/op      123.26 mops/s
Benchmark_FNV1A_____SmallKey    500000000               5.07 ns/op      197.36 mops/s
Benchmark_Meiyan____SmallKey    500000000               4.24 ns/op      236.02 mops/s
Benchmark_Jesteress_SmallKey    500000000               5.32 ns/op      188.08 mops/s
Benchmark_Yorikke___SmallKey    500000000               5.52 ns/op      181.20 mops/s
Benchmark_Bernstein___MedKey    50000000               34.90 ns/op       28.65 mops/s
Benchmark_Murmur3_____MedKey    100000000              17.90 ns/op       55.94 mops/s
Benchmark_FNV1A_______MedKey    50000000               31.90 ns/op       31.37 mops/s
Benchmark_Meiyan______MedKey    200000000               9.28 ns/op      107.76 mops/s
Benchmark_Jesteress___MedKey    200000000               8.15 ns/op      122.65 mops/s
Benchmark_Yorikke_____MedKey    200000000               9.19 ns/op      108.83 mops/s
```

Jesteress

gistfile1.go Go

```go
// Constants for multiples of sizeof(WORD)
const (
	_WSZ    = 4        // 4
	_DWSZ   = _WSZ << 1 // 8
	_DDWSZ  = _WSZ << 2 // 16
	_DDDWSZ = _WSZ << 3 // 32
)

// Jesteress derivative of FNV1A from [http://www.sanmayce.com/Fastest_Hash/]
func Jesteress(data []byte) uint32 {
	h32 := uint32(_OFF32)
	i, dlen := 0, len(data)

	for ; dlen >= _DDWSZ; dlen -= _DDWSZ {
		k1 := *(*uint64)(unsafe.Pointer(&data[i]))
		k2 := *(*uint64)(unsafe.Pointer(&data[i+4]))
		h32 = uint32((uint64(h32) ^ ((k1<<5 | k1>>27) ^ k2)) * _YP32)
		i += _DDWSZ
	}

	// Cases: 0,1,2,3,4,5,6,7
	if (dlen & _DWSZ) > 0 {
		k1 := *(*uint64)(unsafe.Pointer(&data[i]))
		h32 = uint32(uint64(h32)^k1) * _YP32
		i += _DWSZ
	}
	if (dlen & _WSZ) > 0 {
		k1 := *(*uint32)(unsafe.Pointer(&data[i]))
		h32 = (h32 ^ k1) * _YP32
		i += _WSZ
	}
	if (dlen & 1) > 0 {
		h32 = (h32 ^ uint32(data[i])) * _YP32
	}
	return h32 ^ (h32 >> 16)
}
```

HashMap Comparisons

gistfile1.txt

```
2013 MacbookAir 11" i7 1.7Ghz Haswell
Linux mint15 3.8.0-19

================
Go version go1.2.1
================

Benchmark_GoMap___GetSmallKey   500000000                7.57 ns/op      132.05 mops/s
Benchmark_HashMap_GetSmallKey   100000000               14.30 ns/op       70.08 mops/s
Benchmark_GoMap____GetMedKey    500000000                4.83 ns/op      207.01 mops/s
Benchmark_HashMap__GetMedKey    200000000                9.54 ns/op      104.82 mops/s
Benchmark_GoMap____GetLrgKey    500000000                4.39 ns/op      227.79 mops/s
Benchmark_HashMap__GetLrgKey    100000000               24.50 ns/op       40.77 mops/s

================
Go version go1.2.1
================

Benchmark_GoMap___GetSmallKey   200000000                8.77 ns/op      114.02 mops/s
Benchmark_HashMap_GetSmallKey   100000000               14.80 ns/op       67.53 mops/s
Benchmark_GoMap____GetMedKey    500000000                6.21 ns/op      161.05 mops/s
Benchmark_HashMap__GetMedKey    200000000                9.51 ns/op      105.15 mops/s
Benchmark_GoMap____GetLrgKey    100000000               18.30 ns/op       54.68 mops/s
Benchmark_HashMap__GetLrgKey    100000000               24.80 ns/op       40.36 mops/s

================
Go version go1.0.3
================

Benchmark_GoMap___GetSmallKey   50000000                52.20 ns/op       19.17 mops/s
Benchmark_HashMap_GetSmallKey   100000000               15.50 ns/op       64.34 mops/s
Benchmark_GoMap____GetMedKey    50000000                61.60 ns/op       16.24 mops/s
Benchmark_HashMap__GetMedKey    200000000                8.91 ns/op      112.20 mops/s
Benchmark_GoMap____GetLrgKey    20000000                86.90 ns/op       11.51 mops/s
Benchmark_HashMap__GetLrgKey    100000000               25.40 ns/op       39.44 mops/s
```

Some Lessons Learned

- Use `go tool pprof` (linux)
- Avoid short lived objects on the heap
- Use the **stack** or make long lived objects
- Benchmark standard library builtins (strconv)
- Benchmark builtins (defer, hashmap)
- Don’t use channels in performance critical path

Big Lesson Learned?

Go is a good choice
for performance based
systems

Go is getting better faster than the others

Thanks

Resources

- https://github.com/apcera/gnatsd
- https://github.com/apcera/nats
- https://github.com/derekcollison/nats