gohbase

Pure Go HBase Client

Andrey Elenskiy • Bug Breeder at Arista Networks

What’s so special?

- A (sorta)-fully-functional driver for HBase written in Go
- Kinda based on AsyncHBase Java client
- Fast enough
- Small and simple codebase (for now)
- No Java (not a single `AbstractFactoryObserverService`)

Top contributors (2,000 ++)

- Timoha (Andrey Elenskiy)
- tsuna (Benoit Sigoure)
- dgonyeo (Derek Gonyeo)
- CurleySamuel (Sam Curley)

ARISTA

CLOUDFLARE®

So much failure

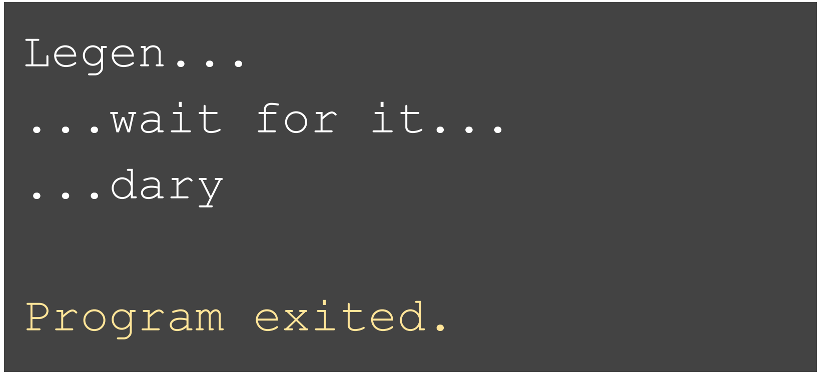
- HBase's feature-set is huge → bunch of small projects → bugs
- Asynch, wannabe lock-free architecture + handling failures = :coding_horror:
- Benchmarking is tricky
- Found some HBase issues in the process

Go is pretty cool I guess

goroutines and channels FTW

```
func main() {
    ch := make(chan string)
    go func() {
        time.Sleep(time.Second)
        ch <- "...wait for it..."
    }()
    go func() {
        time.Sleep(2 * time.Second)
        ch <- "...dary"
    }()

    fmt.Println("Legen...")
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```



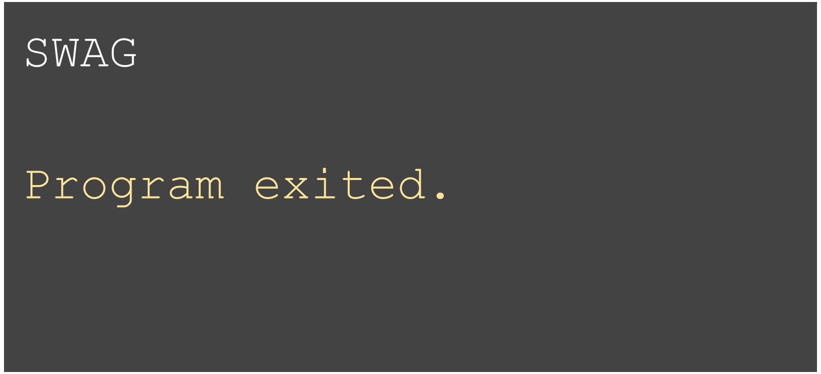
```
Legen...
...wait for it...
...dary

Program exited.
```

context.Context

```go
func main() {
    ch := make(chan string)
    ctx, cancel := context.WithTimeout(context.Background(), 100*time.Millisecond)
    defer cancel()
    go func() {
        time.Sleep(100 * time.Millisecond)
        ch <- "SWAG"
    }()

    // 50/50 chance to fall into either
    select {
    case s := <-ch: // could be HERE
        fmt.Println(s)
    case <-ctx.Done():
        fmt.Println("YOLO") // could or HERE
    }
}
```



```
SWAG


Program exited.
```

Example

```go
func main() {
    client := gohbase.NewClient("localhost")
    // set a timeout for get to be 100 ms
    ctx, cancel := context.WithTimeout(context.Background(), 100*time.Millisecond)
    getRequest, err := hrpc.NewGetStr(ctx, "table", "row")
    // this will fail if it takes longer than 100 ms
    getResponse, err := client.Get(getRequest)
}
```

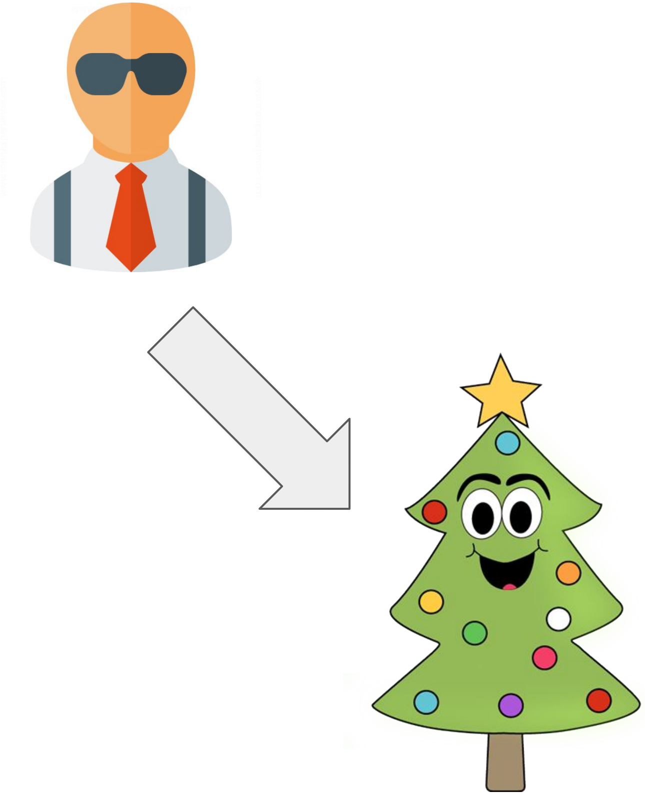
- context is usually used throughout a web app, so it fits to the API nicely
- chaining contexts is useful

```go
func main() {
    parent, cancel := context.WithCancel(context.Background())
    child, _ := context.WithTimeout(parent, 100*365*24*time.Hour) // wait for 100 years
    cancel()
    <-child.Done()
    fmt.Println("YO")
}
```

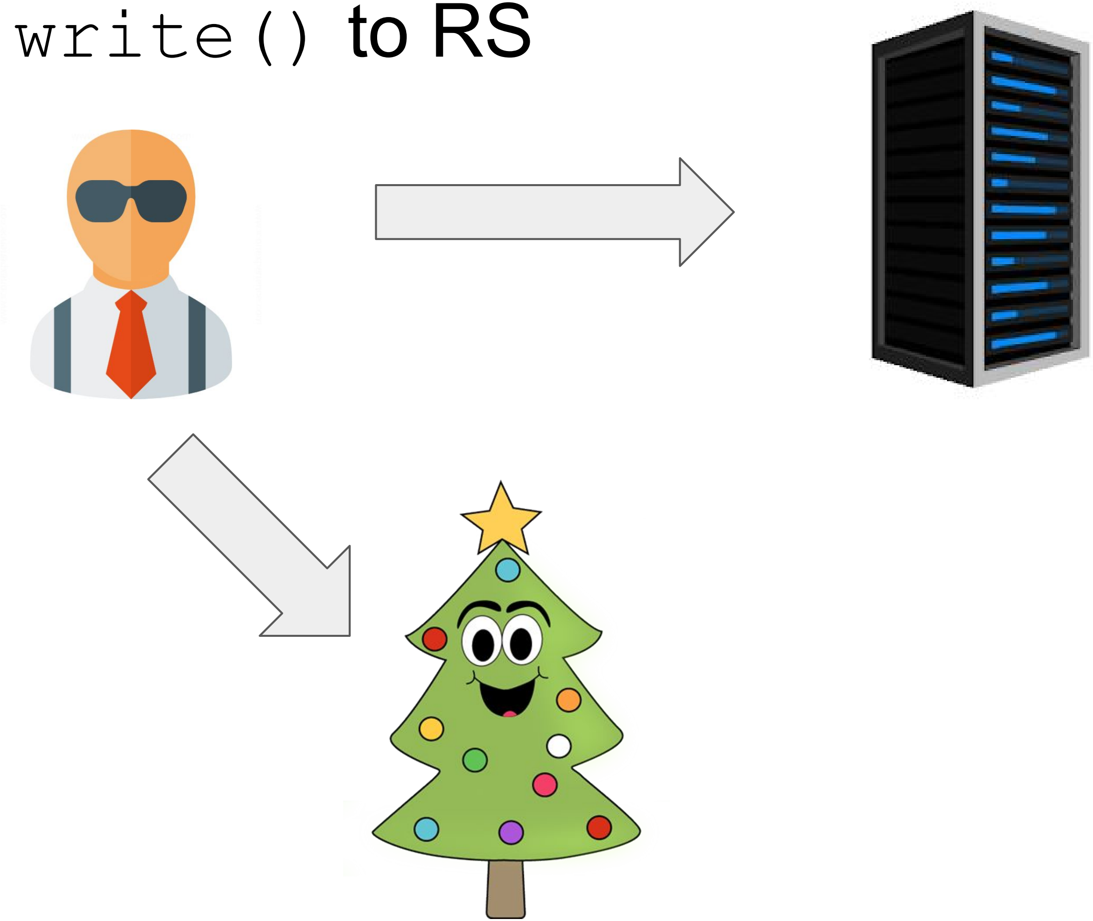
Internal architecture in a nutshell

Case A: Normal (95%)

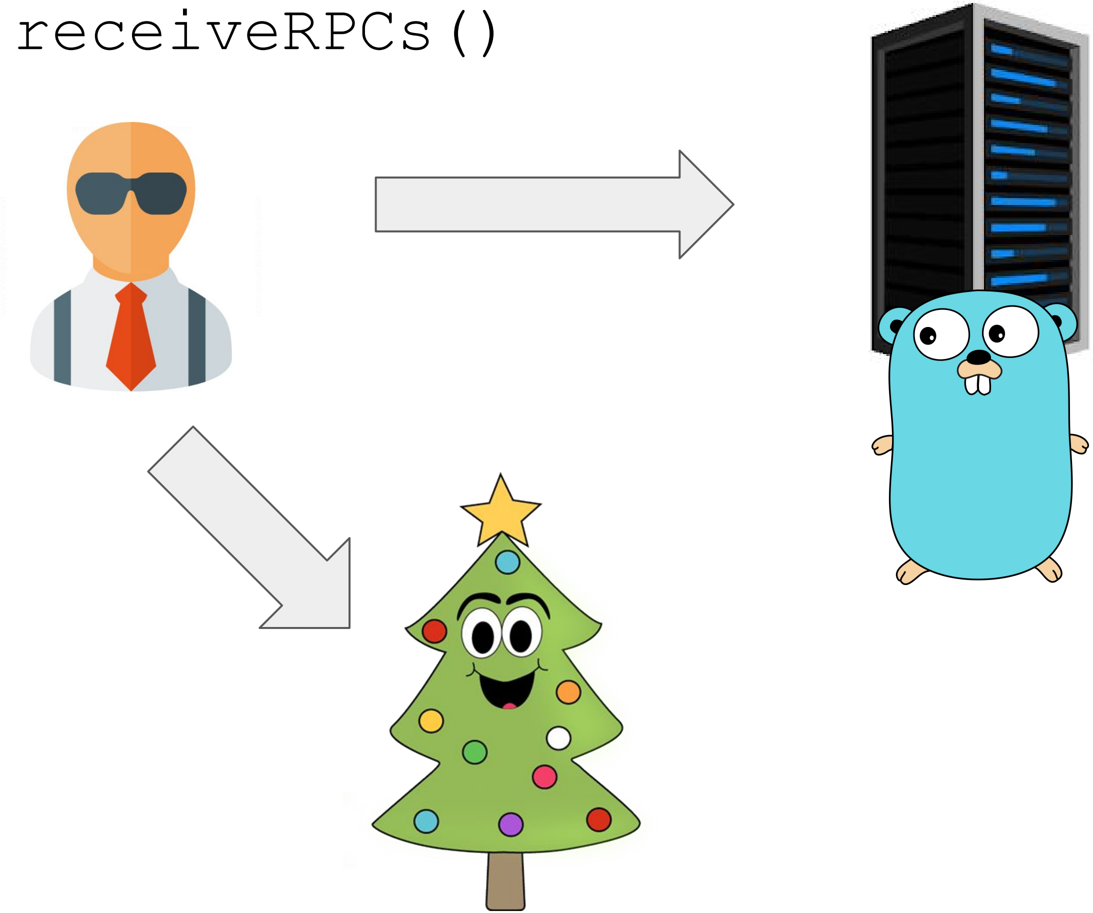
Step 1: Get region in B+Tree



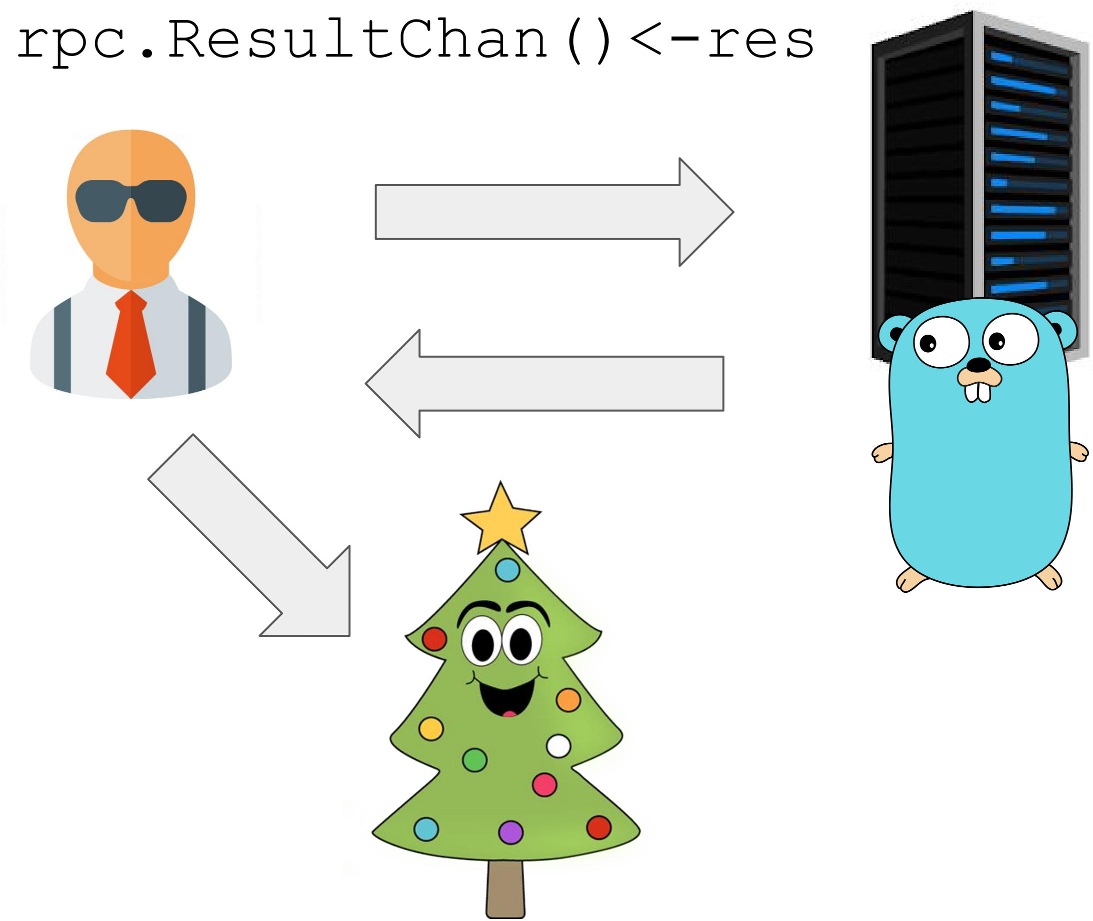
Step 2: `write()` to RS



Step 3: `receiveRPCs()`



Step 4: `rpc.ResultChan()<-res`



Case A: Normal

- Client’s goroutine writes rpc to RS connection
- One goroutine in `RegionClient` to read from RS connection
- *Asynchronous* internals. *Synchronous* API.

Case B: Cache miss/failure

1. Go to B+tree cache for region of the RPC

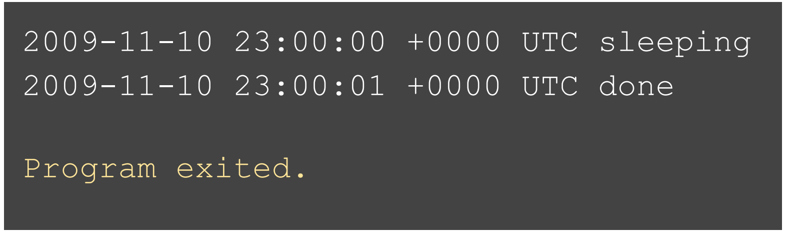
2-100. *...Magic...*

101. `rpc.ResultChan()<-res`

Magic?

2. Mark region as unavailable in cache
3. Block all new RPCs for region by reading on its *"availability"* channel

```
func main() {
    ch := make(chan struct{})
    go func() {
        fmt.Println(time.Now(), "sleeping")
        time.Sleep(time.Second)
        close(ch)
    }()
    <-ch
    fmt.Println(time.Now(), "done")
}
```



```
2009-11-10 23:00:00 +0000 UTC sleeping
2009-11-10 23:00:01 +0000 UTC done

Program exited.
```

4. Start a goroutine to reestablish the region
5. Replace all overlapping regions in cache with new looked up region
6. Connect to Region Server
7. Probe the region to see if it’s being served
8. Close *"availability"* channel to unblock RPCs and let them find new region in cache
9. `write()` to RS

10-100. **PROFIT!!!**

How do you benchmark this stuff?

Requirements:

- No disk IO
- No network

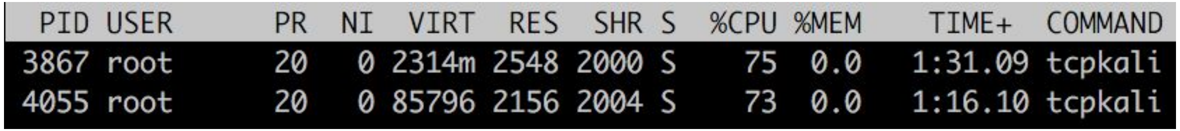
Tried:

- Standalone
- Pseudo-distributed (`MiniHBaseCluster` from `HBaseTestingUtility`)
- Distributed on the same node with Docker
- 16 node HBase cluster

I want my cores

- Using 70% CPU per Region Server on client side
- Region Server is chilling and not using all CPUs per connection
- Where’s the bottleneck?

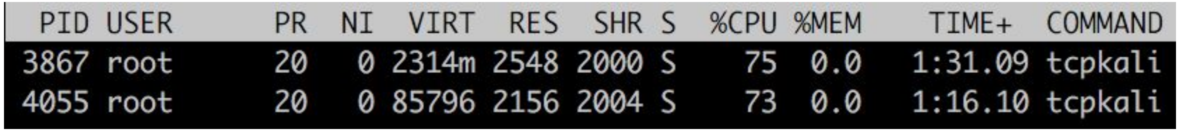
Linux TCP loopback



```
 PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
3867 root      20   0 2314m 2548 2000 S    75  0.0   1:31.09 tcpkali
4055 root      20   0 85796 2156 2004 S    73  0.0   1:16.10 tcpkali
```

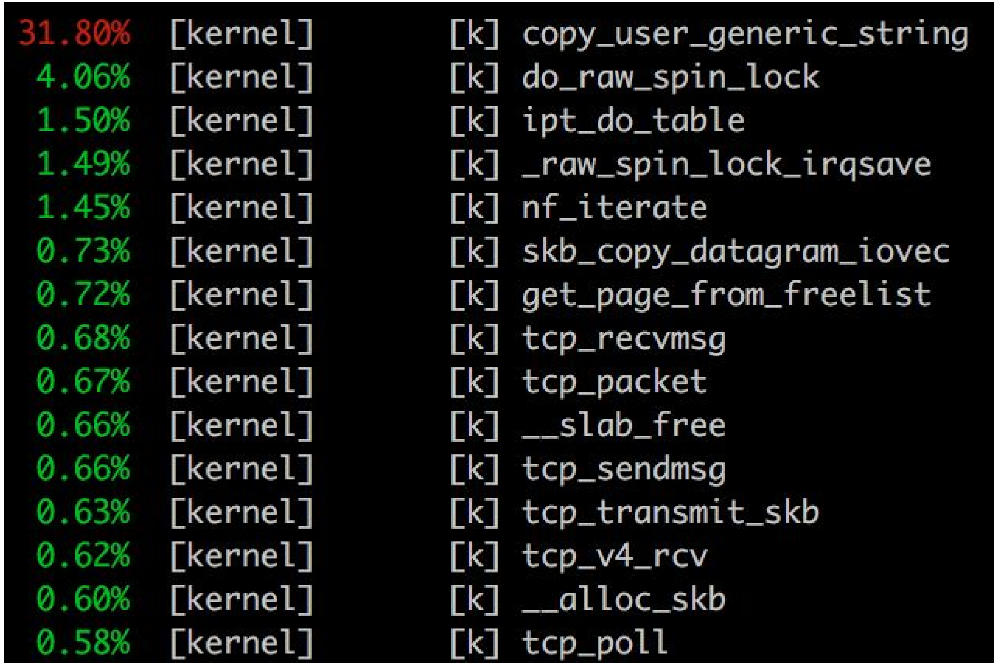
- 100K QPS (10µs per operation)
- 2µs context switch on same hardware
- Where’s ~50% cpu?

Linux TCP loopback



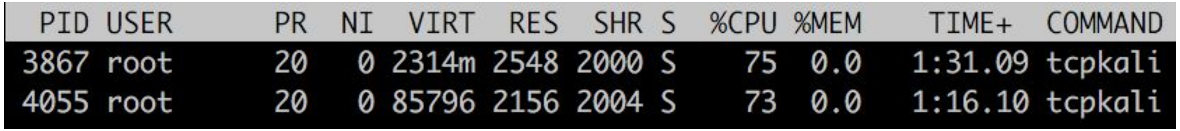
```
  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 3867 root      20   0 2314m 2548 2000 S    75  0.0   1:31.09 tcpkali
 4055 root      20   0 85796 2156 2004 S    73  0.0   1:16.10 tcpkali
```

- 100K QPS (10µs per operation)
- 2µs context switch on same hardware
- Where’s ~50% cpu?



```
31.80%  [kernel]        [k] copy_user_generic_string
 4.06%  [kernel]        [k] do_raw_spin_lock
 1.50%  [kernel]        [k] ipt_do_table
 1.49%  [kernel]        [k] _raw_spin_lock_irqsave
 1.45%  [kernel]        [k] nf_iterate
 0.73%  [kernel]        [k] skb_copy_datagram_iovec
 0.72%  [kernel]        [k] get_page_from_freelist
 0.68%  [kernel]        [k] tcp_recvmsg
 0.67%  [kernel]        [k] tcp_packet
 0.66%  [kernel]        [k] __slab_free
 0.66%  [kernel]        [k] tcp_sendmsg
 0.63%  [kernel]        [k] tcp_transmit_skb
 0.62%  [kernel]        [k] tcp_v4_rcv
 0.60%  [kernel]        [k] __alloc_skb
 0.58%  [kernel]        [k] tcp_poll
```

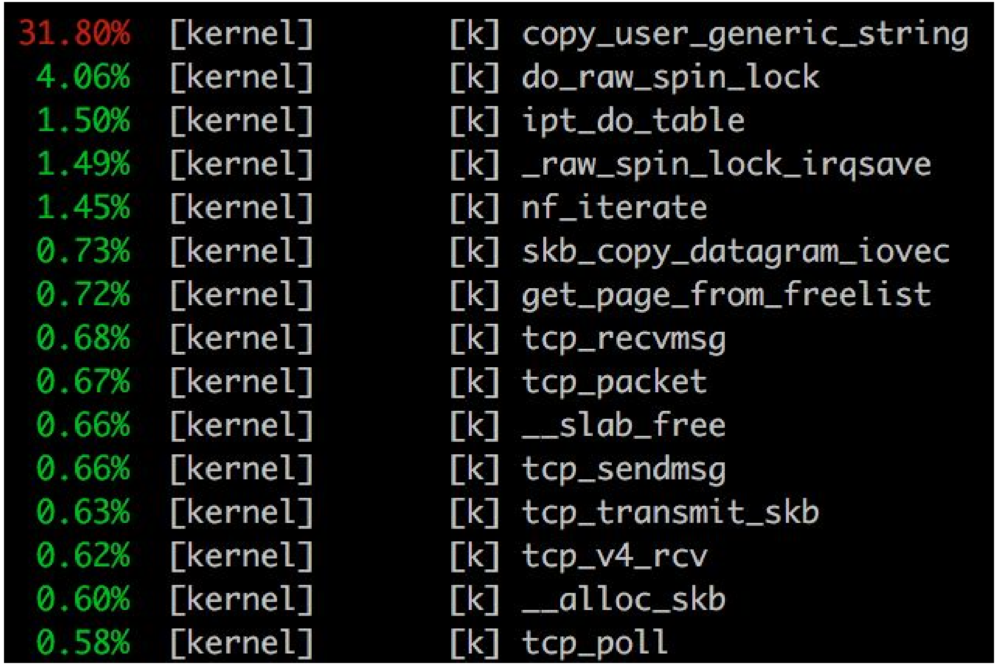
Linux TCP loopback



```
  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 3867 root      20   0 2314m 2548 2000 S    75  0.0   1:31.09 tcpkali
 4055 root      20   0 85796 2156 2004 S    73  0.0   1:16.10 tcpkali
```

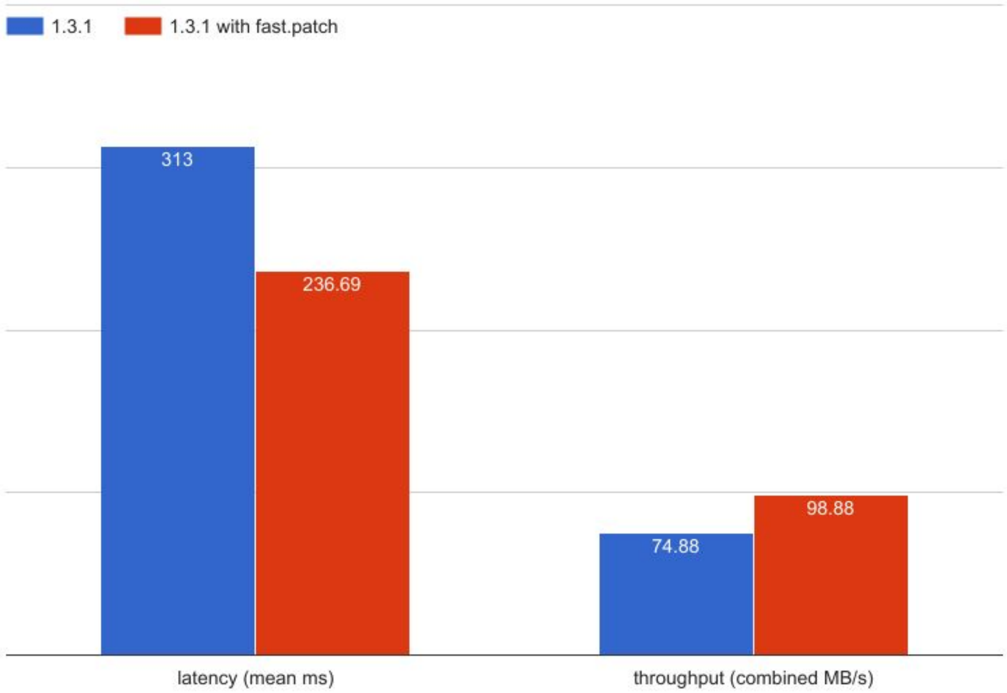
- 100K QPS (10µs per operation)
- 2µs context switch on same hardware
- Where’s ~50% cpu?

- Read/Write syscall is slow? 😮
- One connection, one core
- Gone too deep, I just wanted to benchmark the client…



```
31.80%  [kernel]       [k] copy_user_generic_string
 4.06%  [kernel]       [k] do_raw_spin_lock
 1.50%  [kernel]       [k] ipt_do_table
 1.49%  [kernel]       [k] _raw_spin_lock_irqsave
 1.45%  [kernel]       [k] nf_iterate
 0.73%  [kernel]       [k] skb_copy_datagram_iovec
 0.72%  [kernel]       [k] get_page_from_freelist
 0.68%  [kernel]       [k] tcp_recvmsg
 0.67%  [kernel]       [k] tcp_packet
 0.66%  [kernel]       [k] __slab_free
 0.66%  [kernel]       [k] tcp_sendmsg
 0.63%  [kernel]       [k] tcp_transmit_skb
 0.62%  [kernel]       [k] tcp_v4_rcv
 0.60%  [kernel]       [k] __alloc_skb
 0.58%  [kernel]       [k] tcp_poll
```

fast.patch [HBASE-15594]



24 clients doing 1M random reads each to one HBase 1.3.1 regionserver

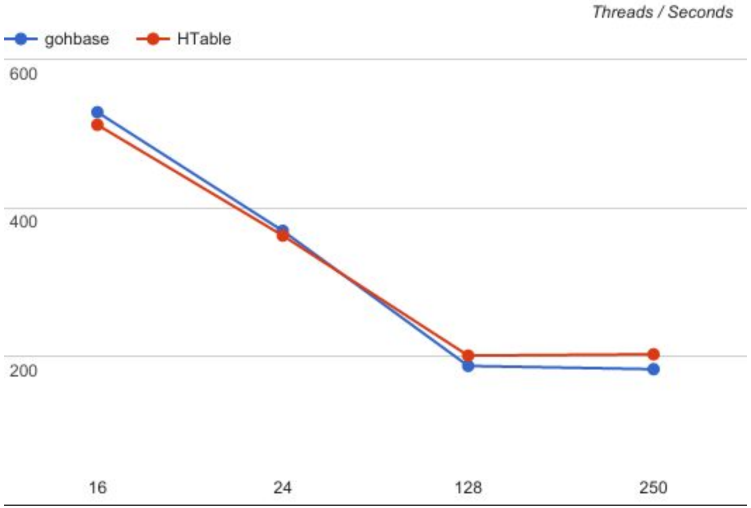
fast.patch [HBASE-15594]

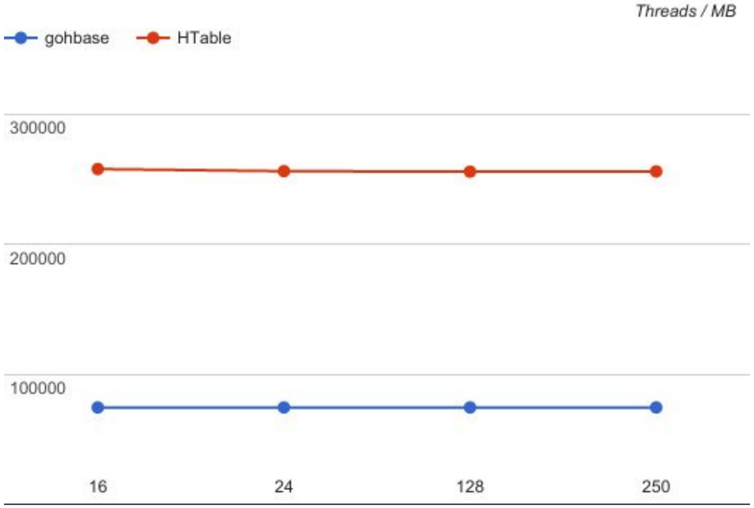
- With it, same %75 / %75 CPU utilization per connection
- Without it, RegionServer is 100% CPU per connection: probably wasting time context switching
- More threads, more throughput
- More connections, even more throughput

Benchmark Results

- 30M rows with 26 byte keys and 100 byte vlaues
- 200 regions
- 3 runs of each benchmark
- 16 regionservers with 32 cores 64gb ram
- One Arista switch ;)

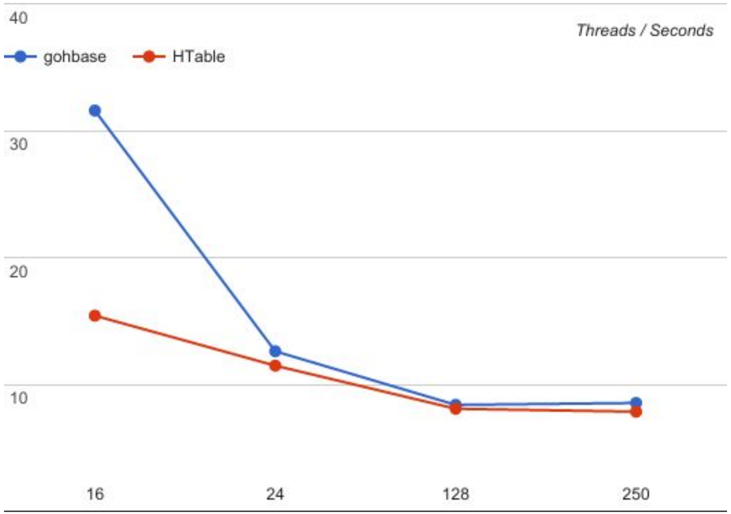
RandomRead

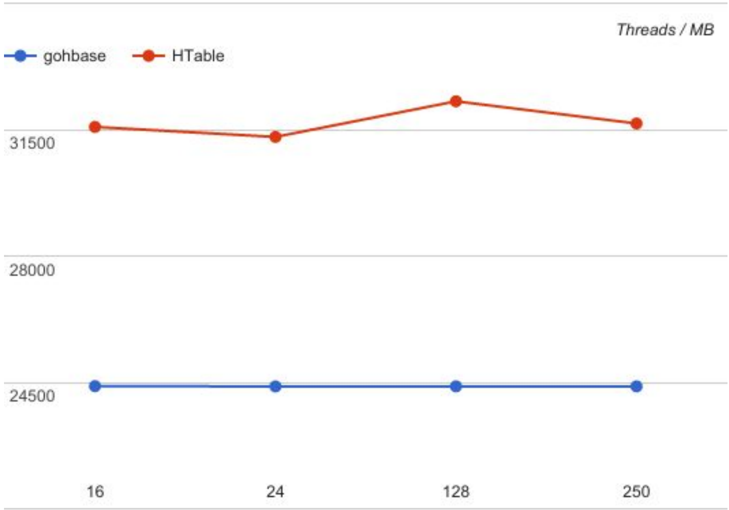




With lots of threads gohbase is **10% faster**
3 times less memory allocated though 🙌

Scan

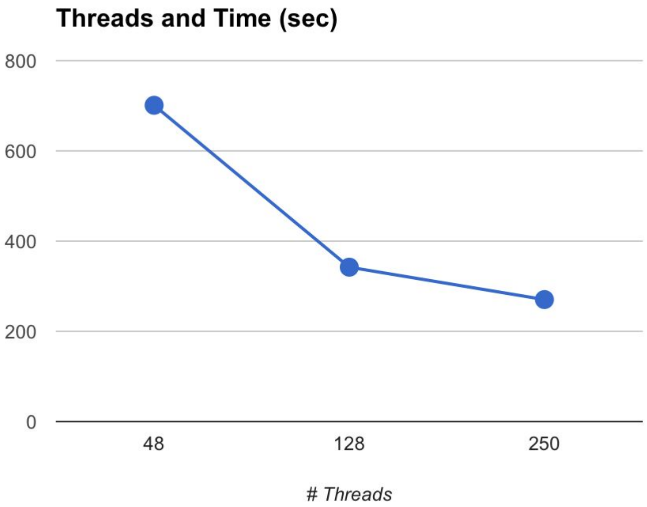


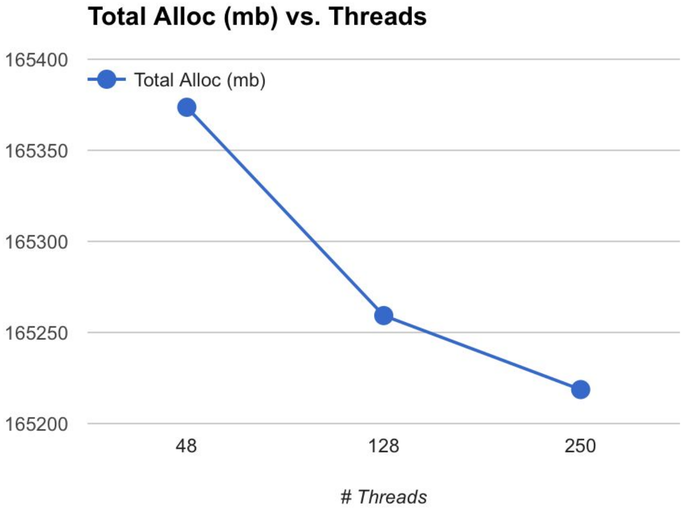


With more threads, gohbase is comparable to HTable

30% less total memory allocated 🙌

RandomWrite? (gohbase only)





Best: 250 threads, 270sec, 165,218mb total

Benchmarking was “entertaining”😫

Region split/merge bug [HBASE-18066]

WTF

Get with `closest_row_before` can return empty cells during a region split/merge

It's not you, it's me

Stayed as skeptical about the bug in HBase as possible, but then gave up and started blaming it

Bug breeding

A bug in gohbase exposed a bug in HBase

What's missing?

- Your usage
- Your contribution
- More unit tests...

Thank you

Andrey Elenskiy
Bug Breeder at Arista Networks
andrey.elenskiy@gmail.com