hgfischer / benchmark+go+nginx.md
Last active 2 days ago

Benchmarking Nginx with Go

benchmark+go+nginx.md

Benchmarking Nginx with Go

There are a lot of ways to serve a Go HTTP application. The best choices depend on each use case. Currently nginx looks to be the standard web server for every new project even though there are other great web servers as well. However, how much is the overhead of serving a Go application behind an nginx server? Do we need some nginx features (vhosts, load balancing, cache, etc) or can you serve directly from Go? If you need nginx, what is the fastest connection mechanism? This are the kind of questions I'm intended to answer here. **The purpose of this benchmark is not to tell that Go is faster or slower than nginx. That would be stupid.**

So, these are the different settings we are going to compare:

- Go HTTP standalone (as the control group)
- Nginx proxy to Go HTTP
- Nginx fastcgi to Go TCP FastCGI
- Nginx fastcgi to Go Unix Socket FastCGI

Hardware

The hardware is a very cheap one. Since we will compare all settings in the same hardware, this should not be a big deal.

- Samsung Laptop NP550P5C-AD1BR
- Intel Core i7 3630QM @2.4GHz (quad core, 8 threads)
- CPU caches: (L1: 256KiB, L2: 1MiB, L3: 6MiB)
- RAM 8GiB DDR3 1600MHz

Software

- Ubuntu 13.10 amd64 Saucy Salamander (updated)
- Nginx 1.4.4 (1.4.4-1~saucy0 amd64)
- Go 1.2 (linux/amd64)
- wrk 3.0.4

Settings

Kernel

Just a little bit tuning in the kernel to allow higher limits. If you have some better idea with this variables, please drop a comment bellow:

```
fs.file-max                     9999999
fs.nr_open                      9999999
net.core.netdev_max_backlog     4096
net.core.rmem_max               16777216
net.core.somaxconn              65535
net.core.wmem_max               16777216
net.ipv4.ip_forward             0
net.ipv4.ip_local_port_range    1025       65535
net.ipv4.tcp_fin_timeout        30
```

```
net.ipv4.tcp_keepalive_time    30
net.ipv4.tcp_max_syn_backlog   20480
net.ipv4.tcp_max_tw_buckets    400000
net.ipv4.tcp_no_metrics_save   1
net.ipv4.tcp_syn_retries       2
net.ipv4.tcp_synack_retries    2
net.ipv4.tcp_tw_recycle        1
net.ipv4.tcp_tw_reuse          1
vm.min_free_kbytes             65536
vm.overcommit_memory           1
```

Limits

The `max open files` limit for `root` and `www-data` was configured to be 200,000.

Nginx

Some nginx tuning is needed. As some folks told me, I disabled `gzip` to be more fair in the comparisons. Here is its config file `/etc/nginx/nginx.conf` :

```
user www-data;
worker_processes auto;
worker_rlimit_nofile 200000;
pid /var/run/nginx.pid;

events {
    worker_connections 10000;
    use epoll;
    multi_accept on;
}

http {
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 300;
    keepalive_requests 10000;
    types_hash_max_size 2048;

    open_file_cache max=200000 inactive=300s;
    open_file_cache_valid 300s;
    open_file_cache_min_uses 2;
    open_file_cache_errors on;

    server_tokens off;
    dav_methods off;

    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    access_log /var/log/nginx/access.log combined;
    error_log /var/log/nginx/error.log warn;

    gzip off;
    gzip_vary off;

    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/sites-enabled/*.conf;
}
```

Nginx vhosts

```
upstream go_http {
    server 127.0.0.1:8080;
    keepalive 300;
}

server {
    listen 80;
    server_name go.http;
    access_log off;
    error_log /dev/null crit;
```

```
    location / {
        proxy_pass http://go_http;
        proxy_http_version 1.1;
        proxy_set_header Connection "";
    }
}

upstream go_fcgi_tcp {
    server 127.0.0.1:9001;
    keepalive 300;
}

server {
    listen 80;
    server_name go.fcgi.tcp;
    access_log off;
    error_log /dev/null crit;

    location / {
        include fastcgi_params;
        fastcgi_keep_conn on;
        fastcgi_pass go_fcgi_tcp;
    }
}

upstream go_fcgi_unix {
    server unix:/tmp/go.sock;
    keepalive 300;
}

server {
    listen 80;
    server_name go.fcgi.unix;
    access_log off;
    error_log /dev/null crit;

    location / {
        include fastcgi_params;
        fastcgi_keep_conn on;
        fastcgi_pass go_fcgi_unix;
    }
}
```

The Go code

```
package main

import (
    "fmt"
    "log"
    "net"
    "net/http"
    "net/http/fcgi"
    "os"
    "os/signal"
    "syscall"
)

var (
    abort bool
)

const (
    SOCK = "/tmp/go.sock"
)

type Server struct {
}

func (s Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    body := "Hello World\n"
    // Try to keep the same amount of headers
    w.Header().Set("Server", "gophr")
    w.Header().Set("Connection", "keep-alive")
    w.Header().Set("Content-Type", "text/plain")
```

```
    w.Header().Set("Content-Length", fmt.Sprint(len(body)))
    fmt.Fprint(w, body)
}

func main() {
    sigchan := make(chan os.Signal, 1)
    signal.Notify(sigchan, os.Interrupt)
    signal.Notify(sigchan, syscall.SIGTERM)

    server := Server{}

    go func() {
        http.Handle("/", server)
        if err := http.ListenAndServe(":8080", nil); err != nil {
            log.Fatal(err)
        }
    }()

    go func() {
        tcp, err := net.Listen("tcp", ":9001")
        if err != nil {
            log.Fatal(err)
        }
        fcgi.Serve(tcp, server)
    }()

    go func() {
        unix, err := net.Listen("unix", SOCK)
        if err != nil {
            log.Fatal(err)
        }
        fcgi.Serve(unix, server)
    }()

    <-sigchan

    if err := os.Remove(SOCK); err != nil {
        log.Fatal(err)
    }
}
```

Check the HTTP headers

To be fair all requests must have the same size.

```
$ curl -sI http://127.0.0.1:8080/
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 12
Content-Type: text/plain
Server: gophr
Date: Sun, 15 Dec 2013 14:59:14 GMT

$ curl -sI http://127.0.0.1:8080/ | wc -c
141



$ curl -sI http://go.http/
HTTP/1.1 200 OK
Server: nginx
Date: Sun, 15 Dec 2013 14:59:31 GMT
Content-Type: text/plain
Content-Length: 12
Connection: keep-alive

$ curl -sI http://go.http/ | wc -c
141



$ curl -sI http://go.fcgi.tcp/
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 12
Connection: keep-alive
```

```
Date: Sun, 15 Dec 2013 14:59:40 GMT
Server: gophr

$ curl -sI http://go.fcgi.tcp/ | wc -c
141



$ curl -sI http://go.fcgi.unix/
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 12
Connection: keep-alive
Date: Sun, 15 Dec 2013 15:00:15 GMT
Server: gophr

$ curl -sI http://go.fcgi.unix/ | wc -c
141
```

Start your engines!

- Configure kernel with sysctl
- Configure nginx
- Configure nginx vhosts
- Load the server as www-data
- Run the benchmarks

The benchmarks

GOMAXPROCS = 1

Go standalone

```
# wrk -t100 -c5000 -d30s http://127.0.0.1:8080/
Running 30s test @ http://127.0.0.1:8080/
  100 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   116.96ms   17.76ms 173.96ms   85.31%
    Req/Sec   429.16     49.20   589.00     69.44%
  1281567 requests in 29.98s, 215.11MB read
Requests/sec:  42745.15
Transfer/sec:      7.17MB
```

Nginx + Go through HTTP

```
# wrk -t100 -c5000 -d30s http://go.http/
Running 30s test @ http://go.http/
  100 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   124.57ms   18.26ms 209.70ms   80.17%
    Req/Sec   406.29     56.94     0.87k    89.41%
  1198450 requests in 29.97s, 201.16MB read
Requests/sec:  39991.57
Transfer/sec:      6.71MB
```

Nginx + Go through FastCGI TCP

```
# wrk -t100 -c5000 -d30s http://go.fcgi.tcp/
Running 30s test @ http://go.fcgi.tcp/
  100 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   514.57ms  119.80ms   1.21s    71.85%
    Req/Sec    97.18     22.56   263.00     79.59%
  287416 requests in 30.00s, 48.24MB read
  Socket errors: connect 0, read 0, write 0, timeout 661
Requests/sec:   9580.75
Transfer/sec:      1.61MB
```

Nginx + Go through FastCGI Unix Socket

```
# wrk -t100 -c5000 -d30s http://go.fcgi.unix/
Running 30s test @ http://go.fcgi.unix/
  100 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   425.64ms   80.53ms 925.03ms   76.88%
    Req/Sec   117.03     22.13   255.00     81.30%
  350162 requests in 30.00s, 58.77MB read
  Socket errors: connect 0, read 0, write 0, timeout 210
Requests/sec:  11670.72
Transfer/sec:      1.96MB
```

GOMAXPROCS = 8

Go standalone

```
# wrk -t100 -c5000 -d30s http://127.0.0.1:8080/
Running 30s test @ http://127.0.0.1:8080/
  100 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    39.25ms    8.49ms  86.45ms   81.39%
    Req/Sec     1.29k   129.27     1.79k    69.23%
  3837995 requests in 29.89s, 644.19MB read
Requests/sec: 128402.88
Transfer/sec:     21.55MB
```

Nginx + Go through HTTP

```
# wrk -t100 -c5000 -d30s http://go.http/
Running 30s test @ http://go.http/
  100 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   336.77ms  297.88ms 632.52ms   60.16%
    Req/Sec     2.36k     2.99k   19.11k    84.83%
  2232068 requests in 29.98s, 374.64MB read
Requests/sec:  74442.91
Transfer/sec:     12.49MB
```

Nginx + Go through FastCGI TCP

```
# wrk -t100 -c5000 -d30s http://go.fcgi.tcp/
Running 30s test @ http://go.fcgi.tcp/
  100 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   217.69ms  121.22ms   1.80s    75.14%
    Req/Sec   263.09    102.78   629.00     62.54%
  721027 requests in 30.01s, 121.02MB read
  Socket errors: connect 0, read 0, write 176, timeout 1343
Requests/sec:  24026.50
Transfer/sec:      4.03MB
```

Nginx + Go through FastCGI Unix Socket

```
# wrk -t100 -c5000 -d30s http://go.fcgi.unix/
Running 30s test @ http://go.fcgi.unix/
  100 threads and 5000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   694.32ms  332.27ms   1.79s    62.13%
    Req/Sec   646.86    669.65     6.11k    87.80%
  909836 requests in 30.00s, 152.71MB read
Requests/sec:  30324.77
Transfer/sec:      5.09MB
```

Conclusions

The first set of benchmarks was made with `ab` and some nginx settings was not very well optimized (gzip was enabled and it was not using keep-alive connections with the Go backend). After changing to `wrk` and following the suggestions to optimize `nginx` the results gone very different.

With GOMAXPROCS=1, the `nginx` overhead is not so big after all, but with GOMAXPROCS=8 the difference is huge. I may try another settings in the future but for now, If you need `nginx` features like virtual hosting, load balancing, caching, etc, use the HTTP proxy connection instead of FastCGI. Some folks said that Go's FastCGI is not well optimized and this may be the cause of the huge differences in the results presented here.

cactus commented on Dec 15, 2013

try enabling backend keepalives in nginx for http proxying. Might help.

```
upstream go-http-backend {
    server 127.0.0.1:8080;
    keepalive 60;
}

server {
    listen 80;
    server_name go.http;
    access_log off;
    error_log /dev/null crit;

    # required for keepalives to be used
    proxy_http_version 1.1;
    proxy_set_header Connection "";

    location / {
        proxy_pass http://go-http-backend;
    }
}
```

ab is also not great in general. wrk is awesome.

azer commented on Dec 15, 2013

See also: https://gist.github.com/azer/5955772

tsenart commented on Dec 15, 2013

To try out results with a constant request rate have a look at http://github.com/tsenart/vegeta

shezarkhani commented on Dec 15, 2013

There might be a typo in the results. Under "GOMAXPROCS = 1" for standalone Go, the avg proc time is higher than max proc time.

mtparet commented on Dec 15, 2013

> ab is also not great in general. wrk is awesome.

julienschmidt commented on Dec 15, 2013

Something more expensive than a "Hello World" would be nice.

And also make some of the requests to static files, I think nginx is a lot faster at handling such requests.
Static files served by Go vs Go + Sendfile + nginx would also be interesting.

hgfischer commented on Dec 15, 2013 Owner

@cactus Thank you. I'll follow your suggestion in the following updates.

@shezarkhani I think it's a ab bug. It was not a typo.

@cactus **@mtparet** I'll add `wrk` tests as well.

@julienschmidt I'll think on something related to the expensive action but the "static file" serving enters in the nginx domain of fancy features that I think it's unnecessary to test, otherwise I would need to duplicate complicated nginx features in Go to be fair.

chrhlnd commented on Dec 16, 2013

Check how many workers in Nginx it is firing up. Make sure you're running 1 per hyper thread like you're doing with go.
server {
worker_processes 8;
worker_rlimit_nofile 200000;
}

divoxx commented on Dec 16, 2013

Also, what about using unix socket as the upstream server instead of http?

metakeule commented on Dec 16, 2013

you should make sure to get no timeout errors in all benchmarks before comparison, because once the connection from nginx to go is broken, the error messages are served really fast.

Barbery commented on Oct 13, 2014

thank you, that is useful data.

jadas commented on Jan 9, 2015

Is it possible to send the traffic over SSL? How to enable persistent support in the tool?

chonthu commented on May 11, 2015

@chrhlnd wondering about your recommendation on setting the worker_processes flag. By default it set to 'aut'o which matches the number of available cpu's on the system. Can you elaborate as to why you would manually adjust this?

rabeesh commented on Aug 23, 2015

@hgfischer Results are same with golang 1.5

taoeffect commented on Nov 1, 2015

@hgfischer what about with `tcp_nodelay` off as per this answer?

pqqplinq commented on Dec 15, 2015

it's really useful resource.Thank you.

Ortall commented on Jan 28

I notice that the benchmark check download file only, is there a benchmark that check file upload (POST/PUT request)?

kartiksura commented on Mar 29

@hgfscher
did you improve the nginx peformance?

I am doing a similar test, I want to explore using nginx as a load balancer over go servers.

My nginx basic server gives ~75k qps (for index.html)
golang server gives ~35k qps

But when I use ngix as load balancer over golang server the throughput drops to ~22k qps

WukongSun commented on May 27

It's so useful.Thank you for your efforts.

khannedy commented 17 days ago

great job!

speaktorob commented 2 days ago

Great effort. Would be interesting to see the results when NGINX and Go and run on separate hardware than wrk.