How to export Prometheus metrics from just about anything

Matt Layher, May 2, 2018

Matt Layher

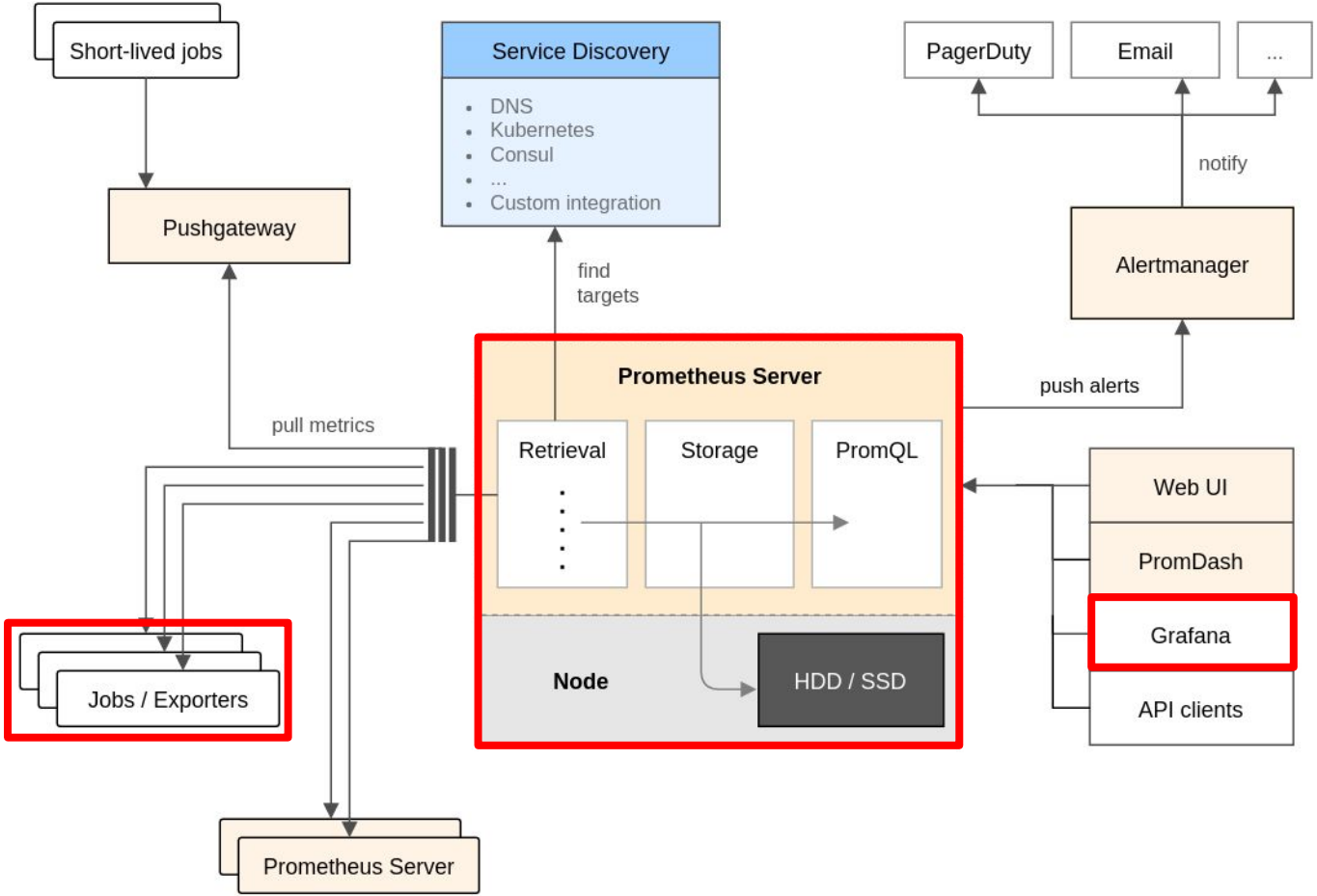
- Senior Engineer at DigitalOcean
- Member of Prometheus team
- GitHub + Twitter: @mdlayher
- github.com/mdlayher/talks

A crash course on Prometheus

What is Prometheus?

- Prometheus is an open-source systems monitoring and alerting toolkit.
- Pull-based metrics gathering system, simple text format for metrics exposition.
- PromQL: powerful query language.

Architecture

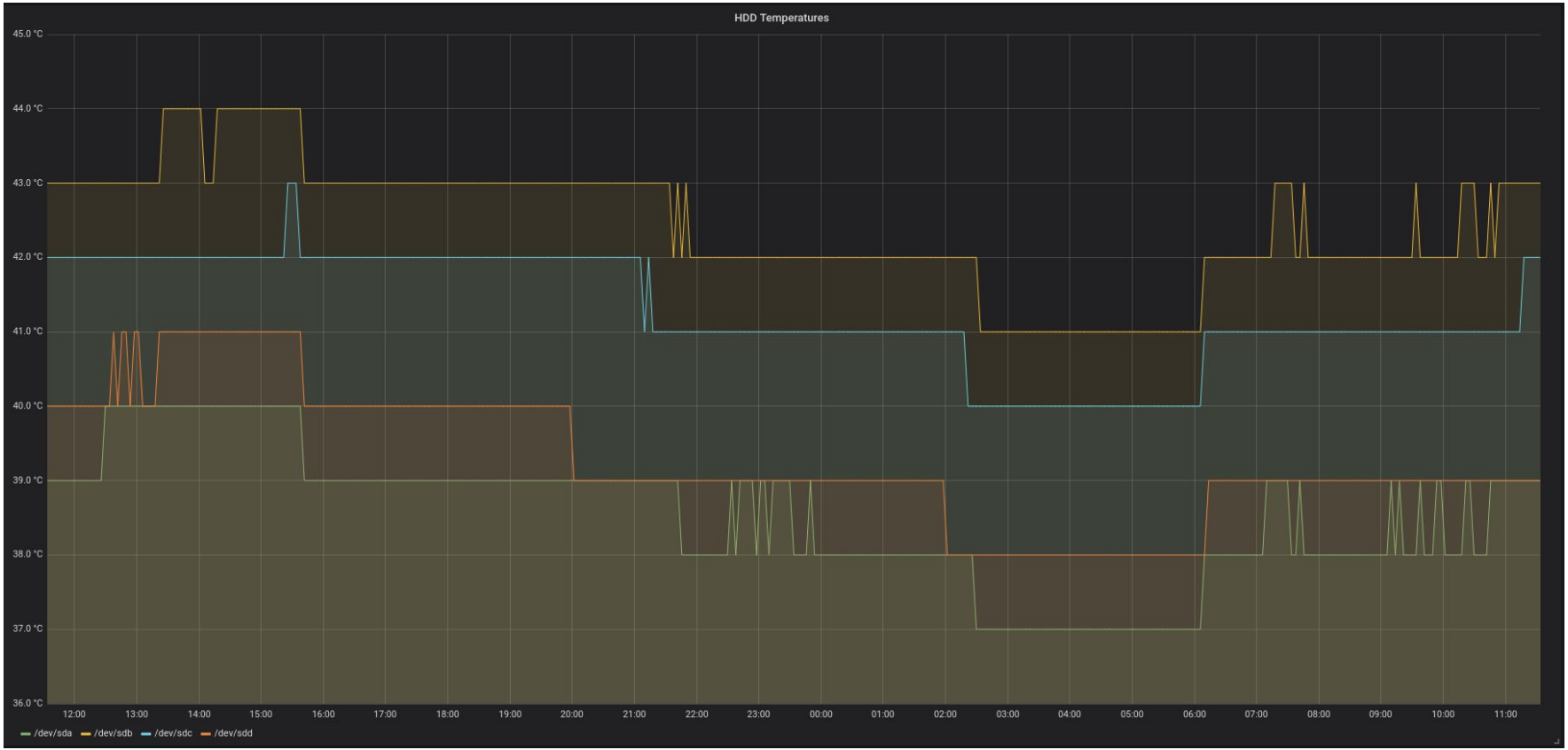


Prometheus text format

```
$ curl -s http://localhost:9100/metrics | grep node_
# HELP node_arp_entries ARP entries by device
# TYPE node_arp_entries gauge
node_arp_entries{device="br0"} 7
# HELP node_boot_time Node boot time, in unixtime.
# TYPE node_boot_time gauge
node_boot_time 1.521387979e+09
# HELP node_context_switches Total number of context switches.
# TYPE node_context_switches counter
node_context_switches 1.55007032e+08
```

PromQL

```
smartmon_temperature_celsius_raw_value{instance="example"}
```



What's a Prometheus exporter?

What's a Prometheus exporter?

- Exporters bridge the gap between Prometheus and systems which don't export metrics in the Prometheus format.

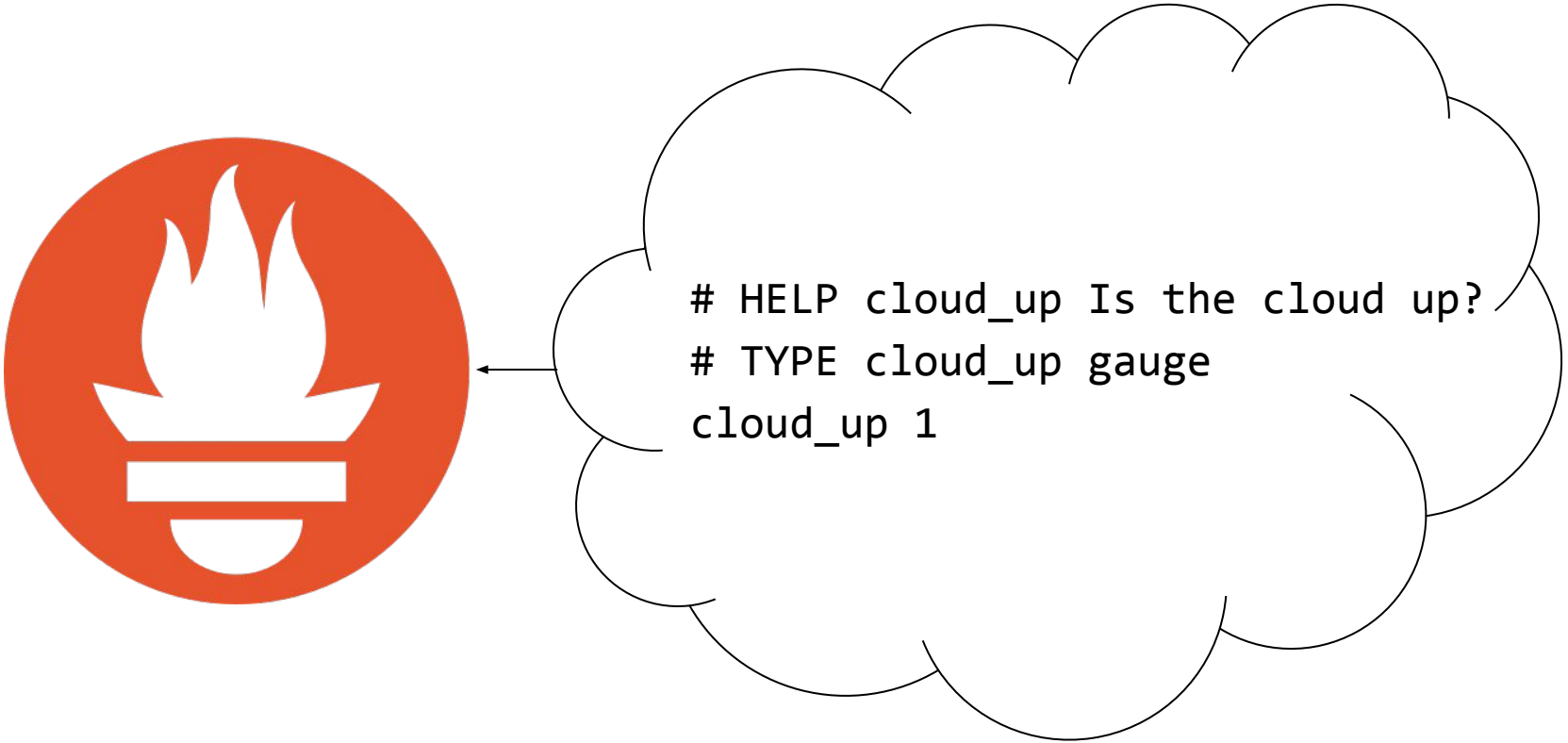
- Typically run on the same machine as a service, but not always!

Example exporters

- node_exporter
  - Exposes system metrics from UNIX-like machines.
- mysqld_exporter
  - Exposes metrics from a MySQL server.
- blackbox_exporter
  - Exposes metrics from “black box” systems via HTTP, ICMP, etc.

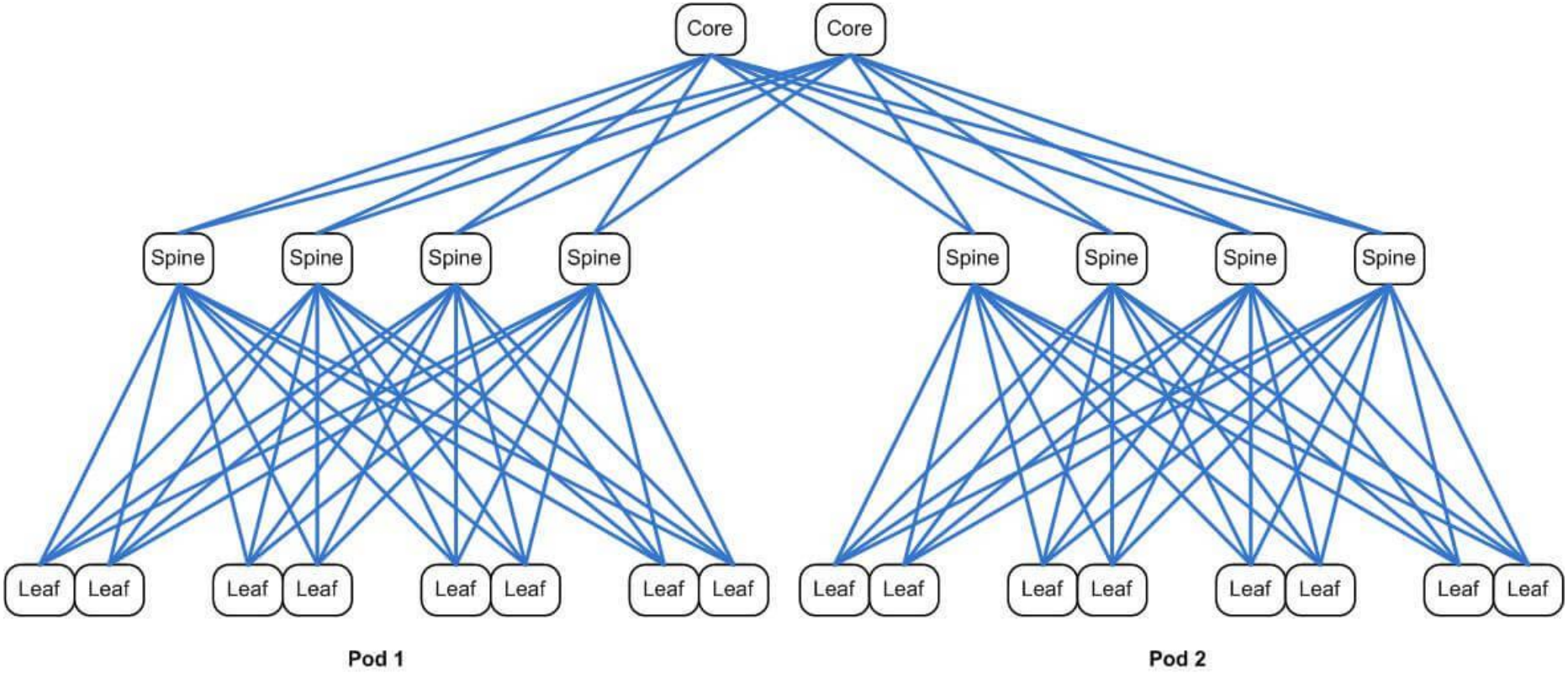
In a cloud-native future...

Cloud-native metrics



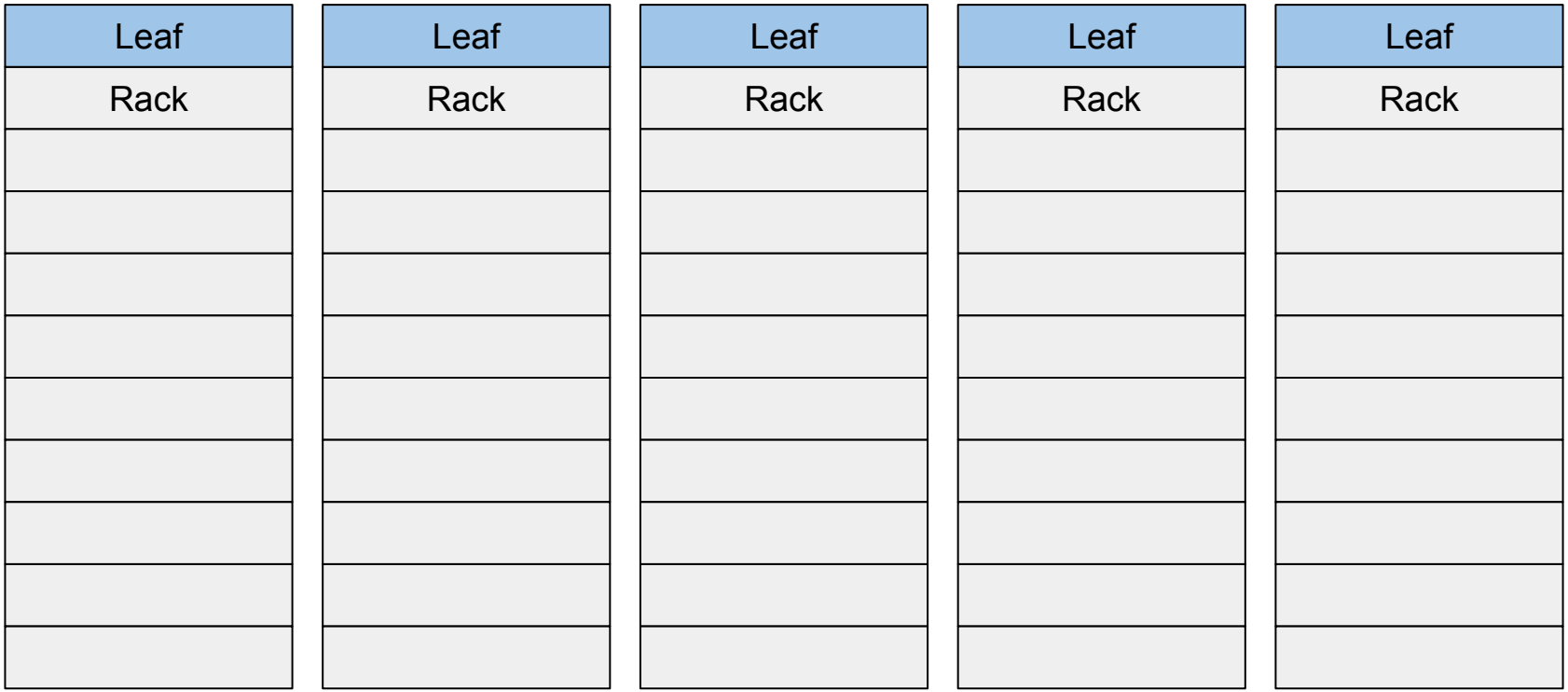
… in a bare metal reality

Bare metal topology



https://blog.digitalocean.com/building-the-next-generation-of-digitalocean-networking/

Bare metal topology



Bare metal metrics…?

```
matt@router# please-give-me-prometheus-metrics
vbash: please-give-me-prometheus-metrics: command not found

matt@server:~$ cat /dev/prometheus-metrics
cat: /dev/prometheus-metrics: No such file or directory
```

Where can I find Prometheus metrics for these systems?

Finding Prometheus exporters

- https://prometheus.io/docs/instrumenting/exporters/
- Search internet, mailing lists, Prometheus wiki, etc.
- … or roll your own!

Basics of building an exporter in Go

Basics of building an exporter in Go

`main` builds types, starts HTTP server

Basics of building an exporter in Go

```
// Make Prometheus client aware of our collector.
c := newCollector(x)
prometheus.MustRegister(c)

// Set up HTTP handler for metrics.
mux := http.NewServeMux()
mux.Handle("/metrics", promhttp.Handler())

// Start listening for HTTP connections.
const addr = ":8888"
log.Printf("starting exporter on %q", addr)
if err := http.ListenAndServe(addr, mux); err != nil {
    log.Fatalf("cannot start exporter: %s", err)
}
```

Basics of building an exporter in Go

`prometheus.Collector` interface

Basics of building an exporter in Go

```
// A collector is a prometheus.Collector for a service.
type collector struct {
    // Possible metric descriptions.
    RequestsTotal *prometheus.Desc

    // A dependency for gathering metrics.
    requests func() (int, error)
}
```

Basics of building an exporter in Go

```
// newCollector constructs a collector.
func newCollector(/* dependencies */) prometheus.Collector {
    return &collector{
        RequestsTotal: prometheus.NewDesc(
            // Name of the metric.
            "exporter_requests_total",
            // The metric's help text.
            "The total number of requests that occur.",
            // The metric's label dimensions.
            nil, nil,
        ),
        requests: /* dependencies */,
    }
}
```

Basics of building an exporter in Go

```
// Describe implements prometheus.Collector.
func (c *collector) Describe(ch chan<- *prometheus.Desc) {
    // Gather metadata about each metric.
    ds := []*prometheus.Desc{
        c.RequestsTotal,
    }

    for _, d := range ds {
        ch <- d
    }
}
```

Basics of building an exporter in Go

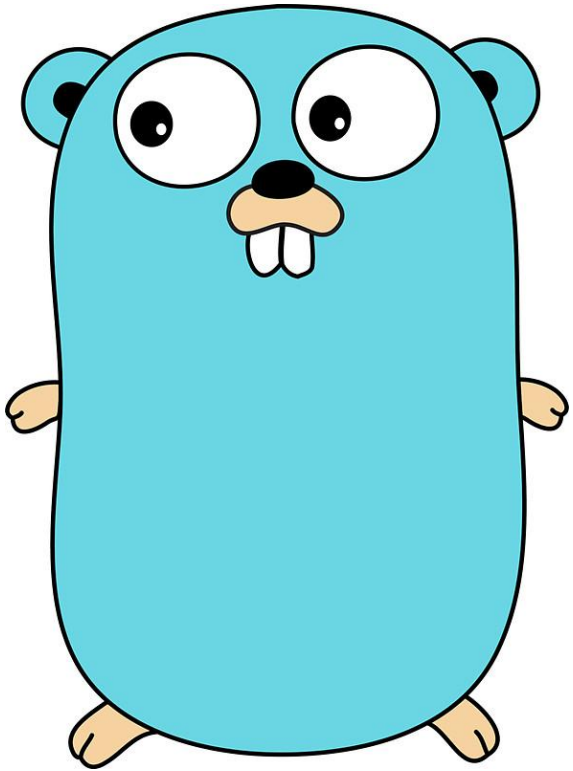
```
// Collect implements prometheus.Collector.
func (c *collector) Collect(ch chan<- prometheus.Metric) {
    // Take a metrics snapshot.  Must be concurrency safe.
    requests, err := c.requests()
    if err != nil {
        // If an error occurs, send an invalid metric to notify
        // Prometheus of the problem.
        ch <- prometheus.NewInvalidMetric(c.RequestsTotal, err)
        return
    }
    // Always use "const metric" constructors.
    ch <- prometheus.MustNewConstMetric(
        c.RequestsTotal, prometheus.CounterValue,
        requests,
    )
}
```

Basics of building an exporter in Go

- Build reusable packages! Don't mix low-level details with exporting metrics!
- Write unit tests! Perform HTTP GET and check that the metrics output is what you expect!
- Make use of `promtool check metrics` for linting!

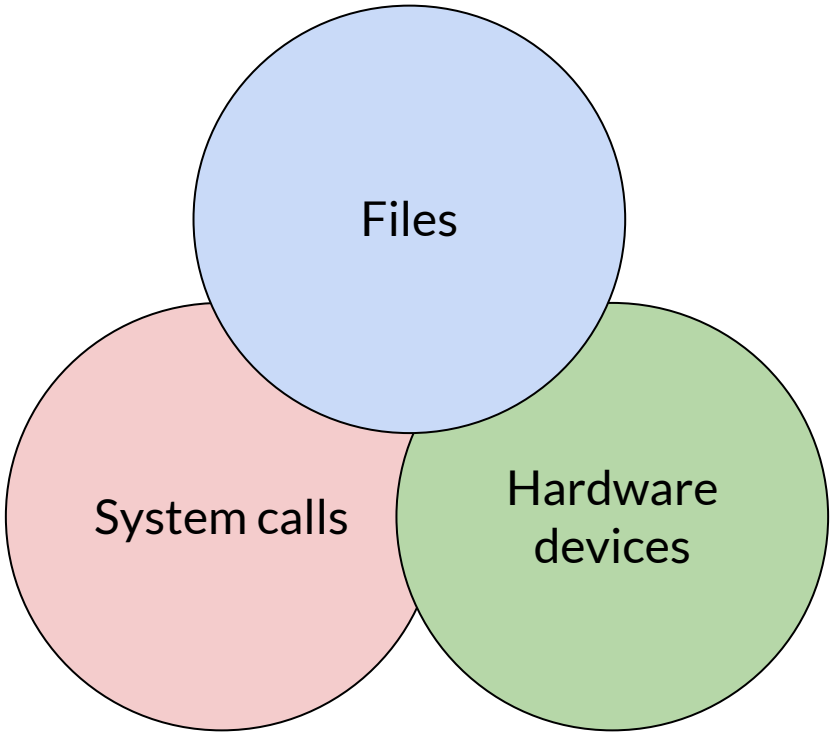
```
$ curl http://localhost:8888/metrics | promtool check metrics
x_gigabytes counter metrics should have "_total" suffix
x_gigabytes use base unit "bytes" instead of "gigabytes"
```

Let's go get some metrics!



Go gopher by Renee French

Sources of metrics



Gathering metrics from files

`/proc/stat`

/proc/stat on Linux

- Kernel/system statistics.
- Numbers indicate amount of time CPU spent in various states like “user”, “system”, “idle”, etc.

```
$ cat /proc/stat | head -n 5
cpu  1203146 8916 341602 12608901 16550 17 2512 0 0 0
cpu0 178422 1228 39617 1538043 2562 6 322 0 0 0
cpu1 178934 1132 40081 1550355 1922 5 94 0 0 0
cpu2 150578 856 65130 1551055 1322 5 1846 0 0 0
cpu3 173647 1264 39402 1557230 1698 0 68 0 0 0
```

Parsing `/proc/stat` in Go

Create a clear and concise exported API

Parsing /proc/stat in Go

```
// A CPUStat contains statistics for an individual CPU.
type CPUStat struct {
    // The ID of the CPU.
    ID string

    // The time, in USER_HZ (typically 1/100th of a second),
    // spent in each of user, system, and idle modes.
    User, System, Idle int
}
```

Parsing /proc/stat in Go

```
// Scan reads and parses CPUStat information from r.
func Scan(r io.Reader) ([]CPUStat, error) {
    s := bufio.NewScanner(r)
    s.Scan() // Skip the first summarized line.

    var stats []CPUStat
    for s.Scan() {
        // ...
    }

    // Be sure to check the error!
    if err := s.Err(); err != nil {
        return nil, err
    }
    return stats, nil
}
```

Parsing `/proc/stat` in Go

Carefully handle slice bounds

Parsing /proc/stat in Go

```
for s.Scan() {
    // Each CPU stats line should have a "cpu" prefix and
    // exactly 11 fields.
    const nFields = 11
    fields := strings.Fields(string(s.Bytes()))
    if len(fields) != nFields {
        continue
    }
    if !strings.HasPrefix(fields[0], "cpu") {
        continue
    }

    // ...
}
```

Parsing /proc/stat in Go

```
// The values we care about (user, system, idle) lie at indices
// 1, 3, and 4, respectively.  Parse these into the array.
var times [3]int
for i, idx := range []int{1, 3, 4} {
    v, err := strconv.Atoi(fields[idx])
    if err != nil {
        return nil, err
    }

    times[i] = v
}

// ...
```

Parsing /proc/stat in Go

```
    // ...

    stats = append(stats, CPUStat{
        // First field is the CPU's ID.
        ID:     fields[0],
        User:   times[0],
        System: times[1],
        Idle:   times[2],
    })
} // End for loop.
```

Parsing `/proc/stat` in Go

Build an example to try out your API

Parsing /proc/stat in Go

```
f, err := os.Open("/proc/stat")
if err != nil {
    log.Fatalf("failed to open /proc/stat: %v", err)
}
defer f.Close()

stats, err := cpustat.Scan(f)
if err != nil {
    log.Fatalf("failed to scan: %v", err)
}

for _, s := range stats {
    fmt.Printf("%4s: user: %06d, system: %06d, idle: %06d\n",
        s.ID, s.User, s.System, s.Idle)
}
```

Parsing /proc/stat in Go

```
$ go build
$ ./cpustat
cpu0: user: 178422, system: 039617, idle: 1538043
cpu1: user: 178934, system: 040081, idle: 1550355
cpu2: user: 150578, system: 065130, idle: 1551055
cpu3: user: 173647, system: 039402, idle: 1557230
```

https://github.com/mdlayher/talks/tree/master/cnceu2018/htepmfjaa/cpustat

Exporting `/proc/stat` metrics

Wire up dependencies in main

Exporting /proc/stat metrics

```
// Called on each collector.Collect.
stats := func() ([]cpustat.CPUStat, error) {
    f, err := os.Open("/proc/stat")
    if err != nil {
        return nil, fmt.Errorf("failed to open: %v", rr)
    }
    defer f.Close()

    return cpustat.Scan(f)
}

// Make Prometheus client aware of our collector.
c := newCollector(stats)
prometheus.MustRegister(c)
```

Exporting /proc/stat metrics

```
// A collector is a prometheus.Collector for Linux CPU stats.
type collector struct {
    // Possible metric descriptions.
    TimeUserHertzTotal *prometheus.Desc

    // A parameterized function used to gather metrics.
    stats func() ([]cpustat.CPUStat, error)
}
```

Exporting `/proc/stat` metrics

Use anonymous structures to simplify code

Exporting /proc/stat metrics

```
stats, err := c.stats()
if err != nil {
    ch <- prometheus.NewInvalidMetric(c.TimeUserHertzTotal, err)
    return
}
for _, s := range stats {
    tuples := []struct {
        mode string
        v    int
    }{
        {mode: "user", v: s.User},
        {mode: "system", v: s.System},
        {mode: "idle", v: s.Idle},
    }
    // ...
}
```

Exporting /proc/stat metrics

```
for _, t := range tuples {
    // prometheus.Collector implementations should always use
    // "const metric" constructors.
    ch <- prometheus.MustNewConstMetric(
        c.TimeUserHertzTotal,
        prometheus.CounterValue,
        float64(t.v),
        s.ID, t.mode,
    )
}
```

Exporting `/proc/stat` metrics

Try your exporter out with `curl`

Exporting /proc/stat metrics

```
$ curl http://localhost:8888/metrics | head -n 5
# HELP cpustat_time_user_hertz_total Time in USER_HZ a given CPU
spent in a given mode.
# TYPE cpustat_time_user_hertz_total counter
cpustat_time_user_hertz_total{cpu="cpu0",mode="idle"}
1.597421e+06
cpustat_time_user_hertz_total{cpu="cpu0",mode="system"} 39621
cpustat_time_user_hertz_total{cpu="cpu0",mode="user"} 160345
```

https://github.com/mdlayher/talks/tree/master/cnceu2018/htepmfjaa/cpustat/cmd/cpustat_exporter

Gathering metrics from files with Go

- Use `io.Reader` interface wherever possible!
- `bufio.Scanner` is your friend!
- Always check slice/array bounds!
- github.com/prometheus/procfs

Gathering metrics from hardware devices

SiliconDust HDHomeRun

HDHomeRun overview

https://www.silicondust.com/product/hdhomerun-prime/

HDHomeRun overview

```
$ hdhomerun_config discover
hdhomerun device 13252C05 found at 192.168.1.8
$ hdhomerun_config 13252C05 get /tuner0/debug
tun: ch=qam:183000000 lock=qam256:183000000 ss=100
snq=100 seq=100 dbg=-381/-5551
dev: bps=38810720 resync=0 overflow=0
cc:  bps=38810720 resync=0 overflow=0
ts:  bps=12514784 te=0 crc=0
net: pps=1192 err=0 stop=0
```

https://www.silicondust.com/support/linux/

HDHomeRun overview

```
$ sudo tcpdump -i eth0 'ip src 192.168.1.8 or ip dst 192.168.1.8'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:39:59.663870 IP 192.168.1.8.65001 > 192.168.1.5.45260: UDP, length 107
12:39:59.673770 IP 192.168.1.5.41701 > 192.168.1.8.65001: Flags [S], seq 1334971662, win 29200, options [mss
1460,sackOK,TS val 1023475 ecr 0,nop,wscale 7], length 0
12:39:59.674000 IP 192.168.1.8.65001 > 192.168.1.5.41701: Flags [S.], seq 1602917205, ack 1334971663, win
4096, options [mss 1460,nop,wscale 0], length 0
12:39:59.674018 IP 192.168.1.5.41701 > 192.168.1.8.65001: Flags [.], ack 1, win 229, length 0
12:39:59.674034 IP 192.168.1.5.41701 > 192.168.1.8.65001: Flags [P.], seq 1:22, ack 1, win 229, length 21
12:39:59.674205 IP 192.168.1.8.65001 > 192.168.1.5.41701: Flags [P.], seq 1:45, ack 22, win 4096, length 44
12:39:59.674213 IP 192.168.1.5.41701 > 192.168.1.8.65001: Flags [.], ack 45, win 229, length 0
^C
7 packets captured
11 packets received by filter
4 packets dropped by kernel
```

https://github.com/Silicondust/libhdhomerun

Gathering HDHomeRun metrics

Build a network client API

Gathering HDHomeRun metrics

```
// A Client is an HDHomeRun client.
type Client struct {
    mu      sync.Mutex
    c       net.Conn
    b       []byte
    timeout time.Duration
}

// Dial dials a TCP connection to an HDHomeRun device.
func Dial(addr string) (*Client, error) {
    conn, err := net.Dial("tcp", addr)
    if err != nil {
        return nil, err
    }
    return NewClient(conn)
}
```

Gathering HDHomeRun metrics

Build low-level communications types

Gathering HDHomeRun metrics

```
// A Packet is used to communicate with HDHomeRun devices.
type Packet struct {
    // Type specifies the type of message this Packet carries.
    Type uint16
    // Tags specifies tags containing optional attributes.
    Tags []Tag
}

// A Tag is an attribute carried by a Packet.
type Tag struct {
    // Type specifies the type of payload this Tag carries.
    Type uint8
    // Data is an arbitrary byte payload.
    Data []byte
}
```

Gathering HDHomeRun metrics

```
// Execute sends a single request to an HDHomeRun device.
func (c *Client) Execute(req *Packet) (*Packet, error) {
    // Serialize all access to the device connection.
    c.mu.Lock()
    defer c.mu.Unlock()

    // Write a single request, read a single response.
    pb, _ := req.MarshalBinary()
    _, _ = c.c.Write(pb)

    n, _ := c.c.Read(c.b)
    var rep Packet
    _ = rep.UnmarshalBinary(c.b[:n])

    return &rep, nil
}
```

Gathering HDHomeRun metrics

Build high-level friendly APIs

Gathering HDHomeRun metrics

```
// Query performs a read-only query to retrieve information.
func (c *Client) Query(query string) ([]byte, error) {
    req := &Packet{
        // https://github.com/xlab/c-for-go
        Type: libhdhomerun.TypeGetsetReq,
        Tags: []Tag{
            {
                Type: libhdhomerun.TagGetsetName,
                Data: strBytes(query)
            },
        },
    }

    rep, _ := c.Execute(req)
    // ... Unpack bytes
}
```

Gathering HDHomeRun metrics

```
// TunerDebug contains debugging information about a TV tuner.
type TunerDebug struct {
    Tuner            *TunerStatus
    Device           *DeviceStatus
    CableCARD        *CableCARDStatus
    TransportStream *TransportStreamStatus
    Network          *NetworkStatus
}
```

Gathering HDHomeRun metrics

```
// TunerDebug retrieves debugging information about a tuner.
func (c *Client) TunerDebug() (*TunerDebug, error) {
    b, err := c.Query("/tuner0/debug")
    if err != nil {
        return "", err
    }

    debug := new(TunerDebug)
    s := bufio.NewScanner(bytes.NewReader(b))
    for s.Scan() {
        // ... parse just like a normal text file!
    }
    return debug, s.Err()
}
```

Gathering HDHomeRun metrics

Build an example to try out your API

Gathering HDHomeRun metrics

```
$ go build
$ ./hdhrctl -n 1 -i 13252c05 /tuner0/debug
tun: ch=qam:183000000 lock=qam256:183000000 ss=100
snq=100 seq=100 dbg=-381/-5551
dev: bps=38810720 resync=0 overflow=0
cc:  bps=38810720 resync=0 overflow=0
ts:  bps=12514784 te=0 crc=0
net: pps=1192 err=0 stop=0
```

https://github.com/mdlayher/hdhomerun

Exporting HDHomeRun metrics

Enable dialing out to remote devices

Exporting HDHomeRun metrics

```
// dial is used to connect to an HDHomeRun device on each
// metrics scrape request.
dial := func(addr string) (*hdhomerun.Client, error) {
    c, err := hdhomerun.Dial(addr)
    if err != nil {
        return nil, err
    }

    // Set timeout to prevent connection leaks!!
    c.SetTimeout(*hdhrTimeout)

    return c, nil
}

// Implements http.Handler.
h := hdhomerunexporter.NewHandler(dial)
```

Exporting HDHomeRun metrics

```
// A handler is an http.Handler that serves Prometheus
// metrics for HDHomeRun devices.
type handler struct {
    dial func(addr string) (*hdhomerun.Client, error)
}

// ServeHTTP implements http.Handler.
func (h *handler) ServeHTTP(w http.ResponseWriter, r
*http.Request) {
    // ...
}
```

Exporting HDHomeRun metrics

```
// Prometheus is configured to send a target parameter
// with each scrape request.
target := r.URL.Query().Get("target")
if target == "" {
    http.Error(w, "missing target", http.StatusBadRequest)
    return
}

host, port, err := net.SplitHostPort(target)
if err != nil {
    // Assume no port was provided and use the default.
    host = target
    port = ":65001"
}
```

Exporting HDHomeRun metrics

```
addr := net.JoinHostPort(host, port)
c, err := h.dial(addr)
if err != nil {
    http.Error(
        w,
        fmt.Sprintf("failed to dial %q: %v", addr, err),
        http.StatusInternalServerError,
    )
    return
}
defer c.Close()

// Wrap raw client for testing, serve metrics for it.
metrics := serveMetrics(c)
metrics.ServeHTTP(w, r)
```

Exporting HDHomeRun metrics

Build an interface around the Client

Exporting HDHomeRun metrics

```
// A device is a wrapper for an HDHomeRun device.
//
// *hdhomerun.Client implements device.
type device interface {
    TunerDebug() (*hdhomerun.TunerDebug, error)
}
```

Exporting HDHomeRun metrics

Try your exporter out with `curl`

Exporting HDHomeRun metrics

```
$ curl 'http://localhost:9137/metrics?target=192.168.1.8'
# HELP hdhomerun_network_packets_per_second Number of packets
per second being sent by the device for this tuner.
# TYPE hdhomerun_network_packets_per_second gauge
hdhomerun_network_packets_per_second{tuner="0"} 839
# HELP hdhomerun_tuner_info Metadata about each of the tuners
available to a device.
# TYPE hdhomerun_tuner_info gauge
hdhomerun_tuner_info{channel="qam:555000000",lock="qam256:555000
000",tuner="0"} 1
```

https://github.com/mdlayher/hdhomerun_exporter

Exporting HDHomeRun metrics

Make Prometheus pass a target parameter

Exporting HDHomeRun metrics

```
scrape_configs:
  - job_name: 'hdhomerun'
    static_configs:
      - targets:
        - '192.168.1.8' # hdhomerun device.
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
      - target_label: __address__
        replacement: '127.0.0.1:9137' # hdhomerun_exporter.
```

https://github.com/prometheus/blackbox_exporter#configuration

Exporting HDHomeRun metrics

hdhomerun (1/1 up)

Endpoint	State	Labels	Last Scrape	Error
http://192.168.1.4:9137/metrics target="192.168.1.8"	UP	instance="192.168.1.8"	9.903s ago	

https://www.robustperception.io/tag/relabelling/

Exporting HDHomeRun metrics

Export synthetic info metrics for PromQL

Exporting HDHomeRun metrics

```
// Tuner ID, target channel, and locked channel.
labels := []string{
    tuner,
    ts.Channel,
    ts.Lock,
}

// Synthetic information metric, for use with PromQL.
//
// Adding these labels to all metrics could cause
// cardinality problems!
ch <- prometheus.MustNewConstMetric(
    c.TunerInfo, prometheus.GaugeValue, 1, labels...,
)
```

Exporting HDHomeRun metrics

Use info metrics for powerful queries

Exporting HDHomeRun metrics

- “What is the packets per second on a given channel?”

```
hdhomerun_network_packets_per_second{tuner="0"} *
    on (instance, job) group_left(channel)
    hdhomerun_tuner_info{tuner="0"}

{channel="3",instance="192.168.1.8",job="hdhr",tuner="0"} 840
```

https://www.robustperception.io/how-to-have-labels-for-machine-roles/

Gathering metrics from hardware devices with Go

- Set timeouts for network connections!
- Use interfaces for testing!
- Create synthetic metrics instead of adding a label to all metrics!
- Use the power of relabelling!

Gathering metrics from system calls

`statfs(2)` on Linux

statfs(2) on Linux

- `statfs`, `fstatfs` - get filesystem statistics
- The function `statfs()` returns information about a mounted filesystem.

```
$ ./statfs /
/ (EXT4): 4005888 files
$ ./statfs /srv
/srv (XFS): 485822944 files
$ ./statfs /mnt/nfs/media
/mnt/nfs/media (NFS): 15672334874 files
```

Using `statfs` in Go

Build a high-level, OS agnostic, API

Using statfs in Go

```
// A Filesystem contains statistics about a given filesystem.
type Filesystem struct {
    Path  string
    Type  Type
    Files uint64
}

// Type is the type of filesystem detected.
type Type int

// List of possible filesystem types.
const (
    EXT4 Type = iota
    NFS
    XFS
)
```

Using statfs in Go

```
// Get retrieves stats for the filesystem mounted at path.
func Get(path string) (*Filesystem, error) {
    // Call the OS-specific version of get.
    fs, err := get(path)
    if err != nil {
        return nil, err
    }

    fs.Path = path
    return fs, nil
}
```

Using `statfs` in Go

Syscall must be guarded with build tags

https://go-proverbs.github.io/

Using statfs in Go

```
//+build linux

package statfs

import "golang.org/x/sys/unix"

// ... code in statfs_linux.go
```

Using statfs in Go

```
// get is the Linux implementation of get.
func get(path string) (*Filesystem, error) {
    // Structure is populated by passing a pointer to it.
    var s unix.Statfs_t
    if err := unix.Statfs(path, &s); err != nil {
        return nil, err
    }

    // Return the non-OS-specific structure.
    return &Filesystem{
        Type:  linuxType(s.Type),
        Files: s.Files,
    }, nil
}
```

Using statfs in Go

```
//+build !linux

// ... code in statfs_others.go
package statfs

import (
    "fmt"
    "runtime"
)

// get is unimplemented.
func get(_ string) (*Filesystem, error) {
    return nil, fmt.Errorf("statfs not implemented on %s",
runtime.GOOS)
}
```

Using `statfs` in Go

Build an example to try out your API

Using statfs in Go

```
func main() {
    flag.Parse()
    path := flag.Arg(0)
    if path == "" {
        fmt.Println("usage: statfs [path]")
        return
    }

    fs, err := statfs.Get(path)
    if err != nil {
        log.Fatalf("failed to get filesystem: %v", err)
    }

    fmt.Printf("%s (%s): %d files\n", fs.Path, fs.Type,
fs.Files)
}
```

Using statfs in Go

```
$ go build
$ ./statfs /
/ (EXT4): 4005888 files
$ ./statfs /srv
/srv (XFS): 485822944 files
$ ./statfs /mnt/nfs/media
/mnt/nfs/media (NFS): 15672334874 files
```

https://github.com/mdlayher/talks/tree/master/cnceu2018/htepmfjaa/statfs

Exporting `statfs` metrics

… an exercise for the reader!

Gathering metrics from system calls with Go

- Build a high-level, easy to use API!
- Always use build tags with syscalls!
- Be cautious using elevated privileges!

Conclusions

Typical software best practices

Conclusions

- Avoid global/package state by passing dependencies as parameters!
- Create simple, focused, re-usable package APIs when building exporters!
- Read up on Prometheus metrics best practices and apply them judiciously!

Thanks!

mdlayher@gmail.com
github.com/mdlayher
twitter.com/mdlayher