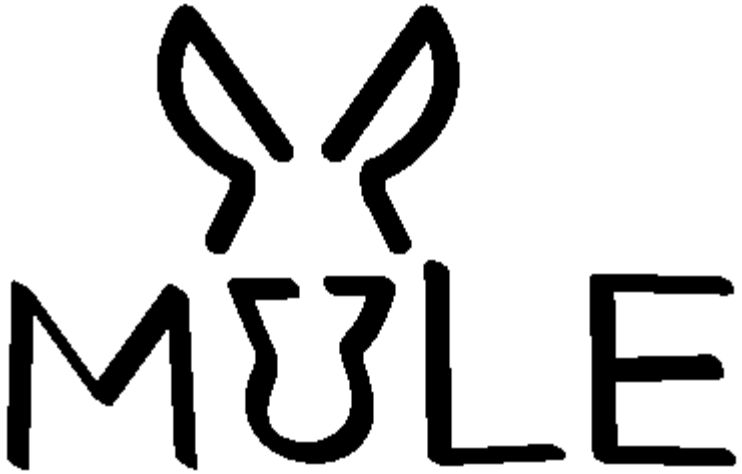
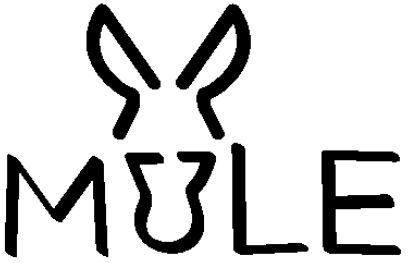


An Application Management Tool in Go

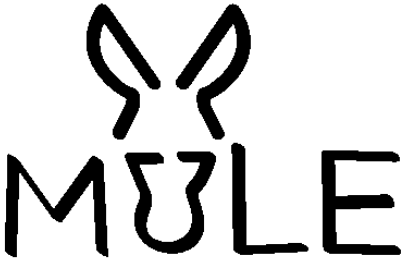
Marc Sauter, René Zbinden PostFinance

A Brief History of Application Management Tools @PostFinance



- C -> Perl (TUI)
- TCL (TUI)
- Java (Agent, Backend, GUI)
  - Performance problems -> (very expensive) refactoring -> still problems -> project terminated
- Revival of Perl due to the lack of alternatives
- Mule … since 2016 and still no need to replace it!
  - the first tool where the stakeholders were able to configure it

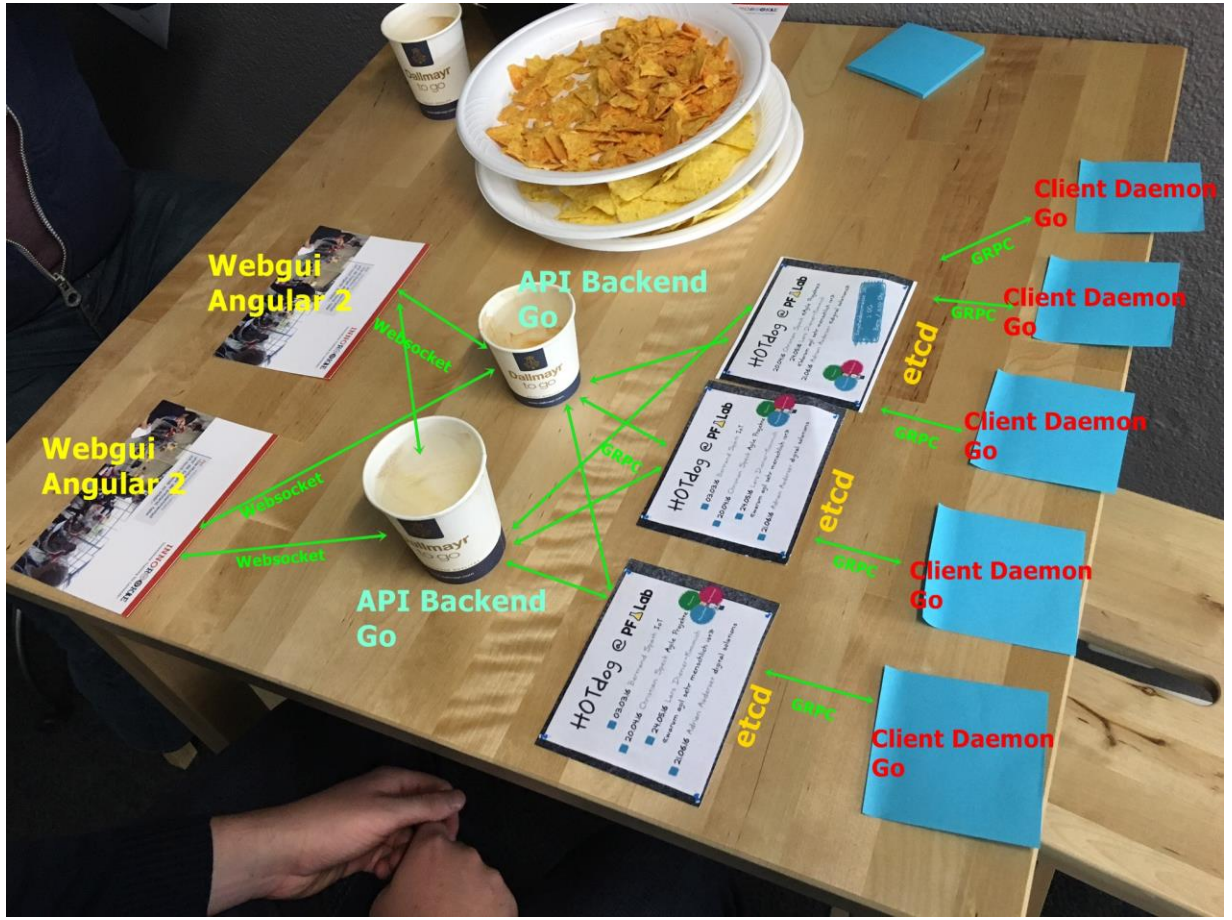
Go@PostFinance



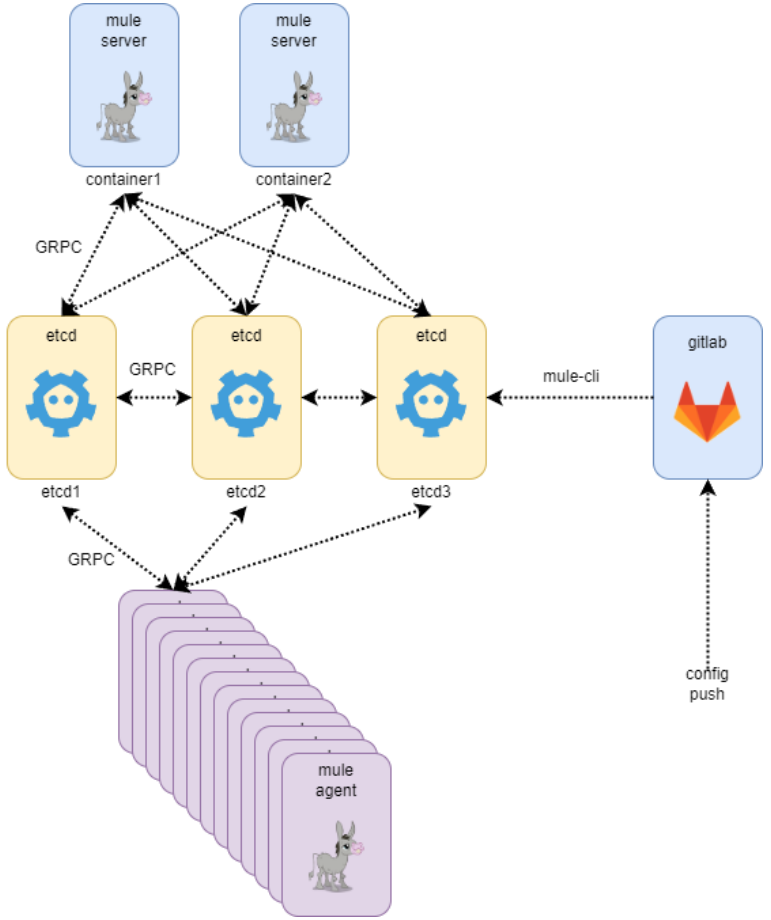
- 2014 (maybe even 2012) First command line tools
- 2015 Certificate Enrollment System
  - Agent for Linux, Solaris and Windows
- 2016 Mule
  - Backend on Kubernetes
  - Agent for Linux, Solaris and Windows
- Today Go is the first choice for infrastructure tools and IaC.
  - PostFinance uses mainly Java and Oracle for application development.

First Design - 21.04.2016

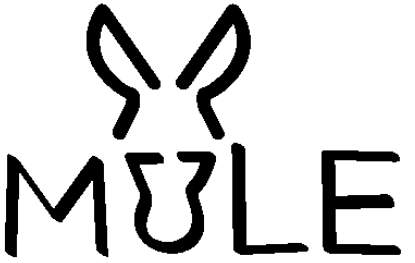
MULE



Architecture

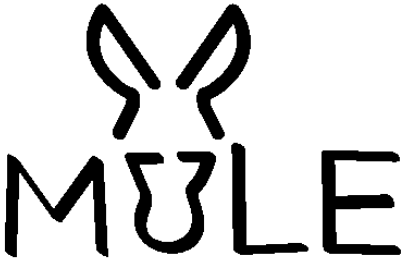


Components



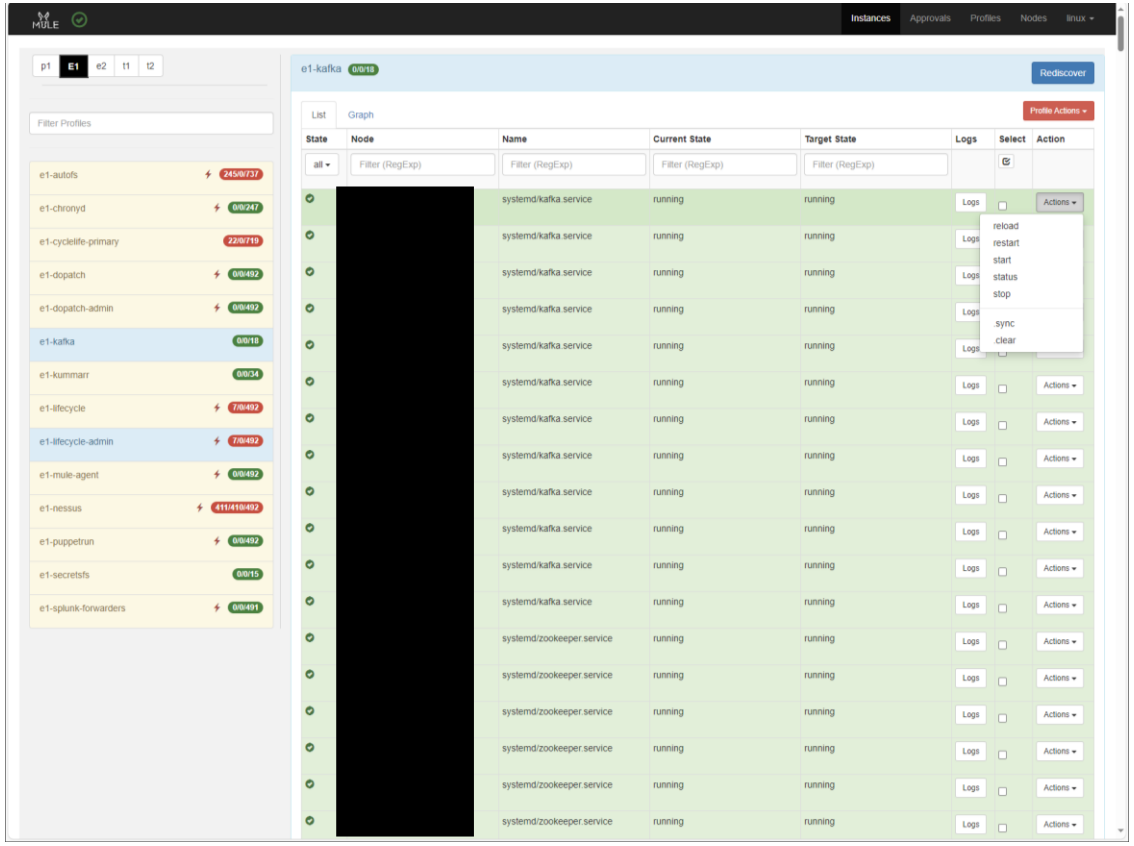
- Backend
  - GUI-Server (REST-API)
  - Go
- Agent
  - Go
- GUI
  - Single Page Application with ReactJS
- Store
  - etcd
- Configuration
  - YAML files in a Git repository … YAML was a well known format for the stakeholders

Principles

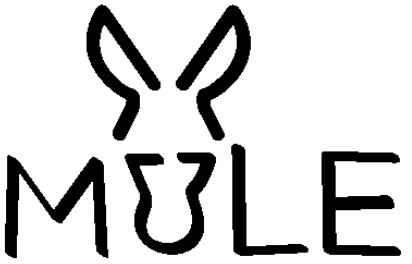


- Configuration in YAML
  - Easy to read/edit
  - Self Service
- Applications and Services get discovered and registered automatically
- The runtime state can be recovered from the configuration
  - etcd backup is not necessary
- At most one writer per etcd key to prevent concurrency issues
- The stakeholders are part of the project

GUI – developed by the stakeholders



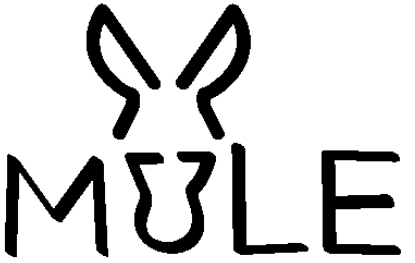
Launch



- It worked much better than we expected.
- It scaled to > 2k agents.
- But we still had to make a few pprof flamegraphs to fix some performance issues.
- Many performance issues in the beginning were related to etcd. Most issues could be fixed with code optimization, some were fixed with etcd upgrades.
- Added the etcd-proxy to reduce the number of connections to the etcd itself.

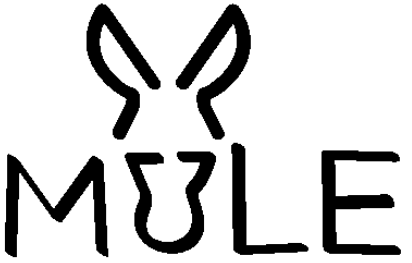
Go was the right choice!

Problems



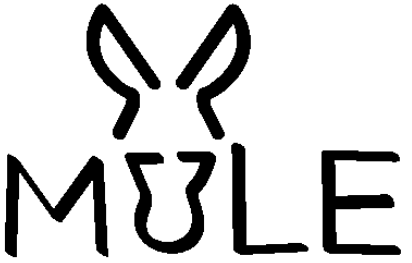
- Code Quality
  - After 5 years of extending and bugfixing, the code was not in good shape anymore.
  - Mule is a kind of hobby and not our main job.
- ETCD
  - Etcd was the core of Mule. Every server and agent instance was connected to etcd and had multiple watchers running.
  - Etcd was used as a runtime store and (misused as) a message queue.
  - > 2k active clients
  - > 4k watchers
  - very sensitive to network and/or disk I/O latency

Refactoring

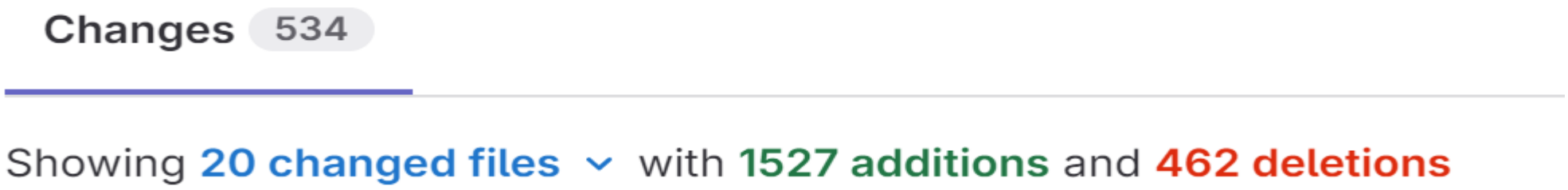


1. Refactor code without changing the architecture
2. Review Architecture
3. Refactor code with new architecture

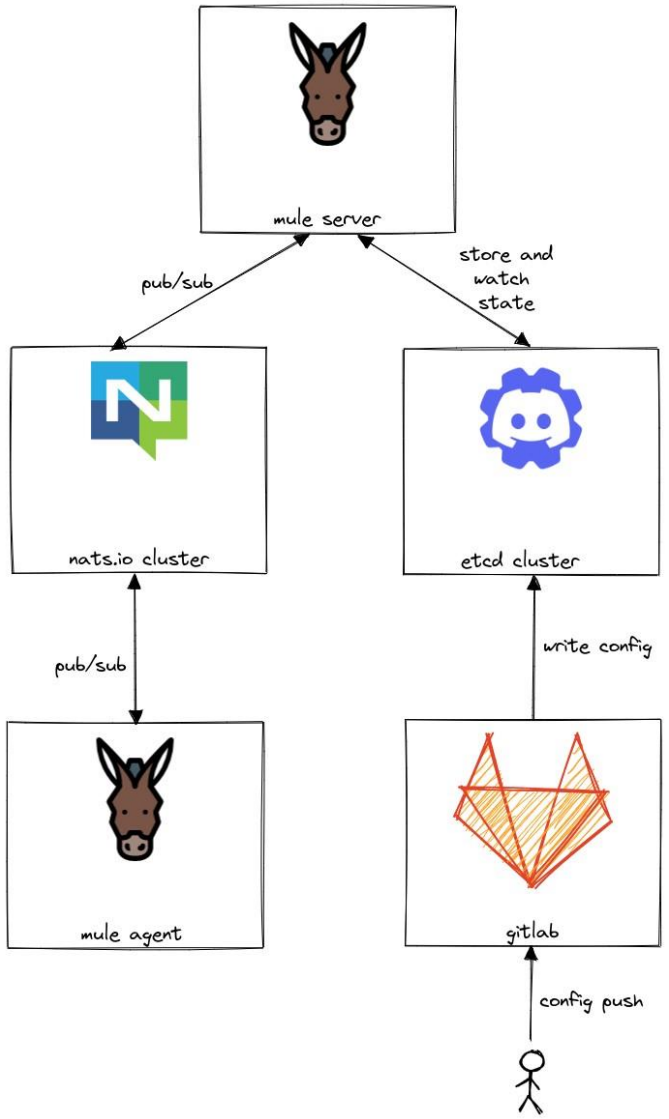
Refactoring – Keep Architecture



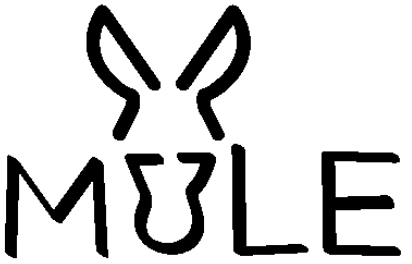
- Configure `golangci-lint`
- Rewrite discovery service
- Improve Observability
- Replace
  - `pkg/errors` with `errors` from stdlib
  - custom logger implementation with `uber-go/zap`



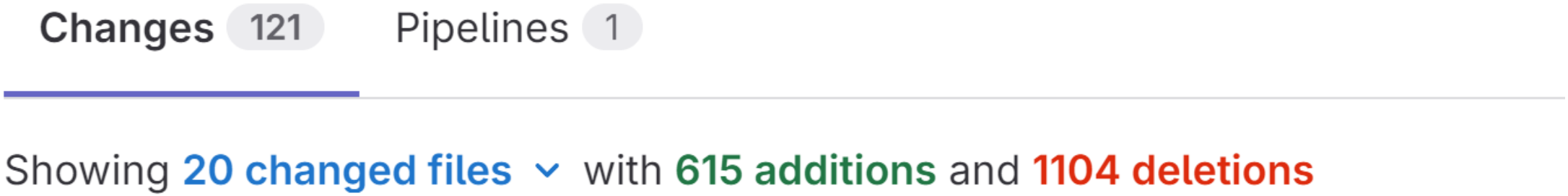
NATS - Architecture



NATS

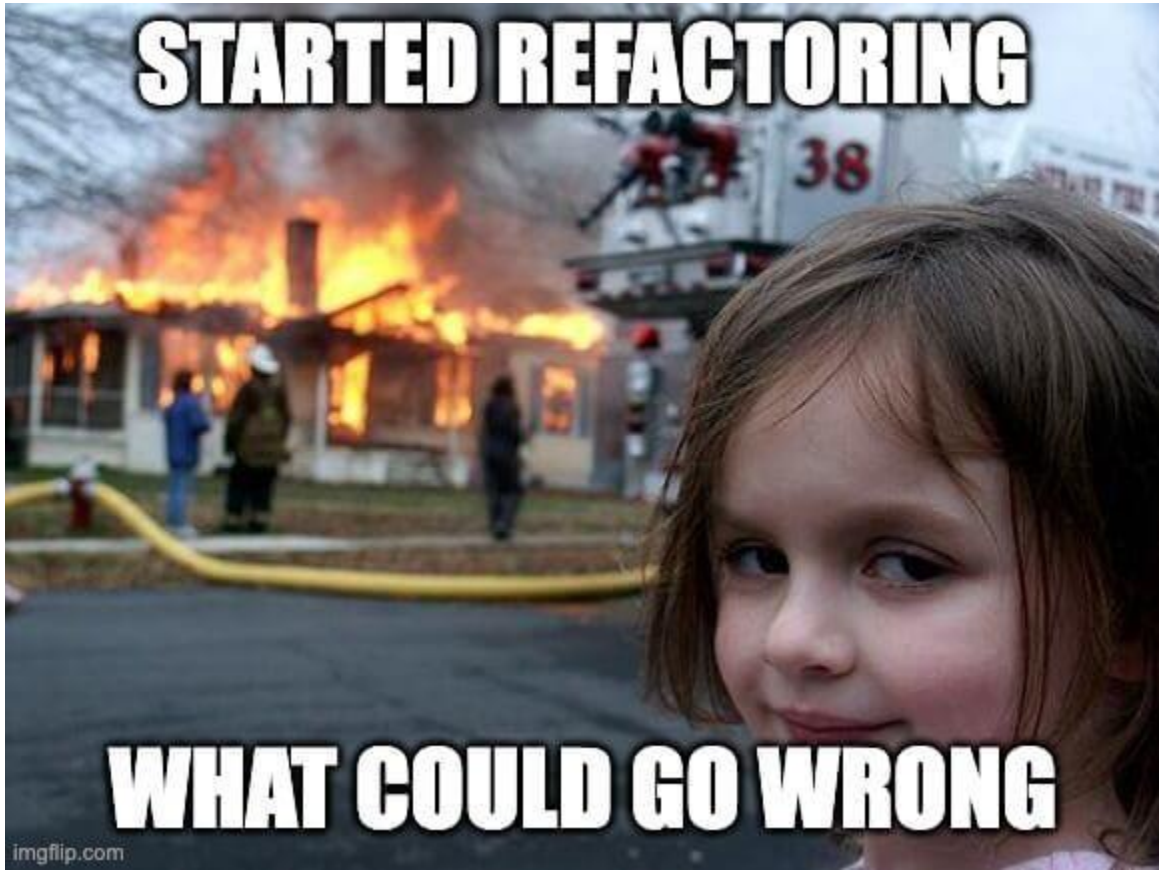


- Released April 2022

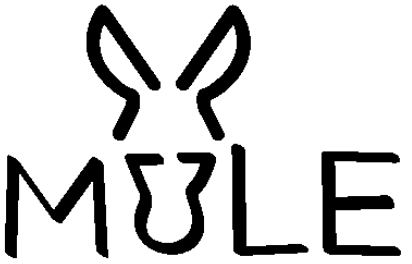


NATS

MULE



Several «Small» Bugs



feat: better approval validation •••
René Zbinden authored 1 year ago

fix: only debug log when no output found •••
René Zbinden authored 1 year ago

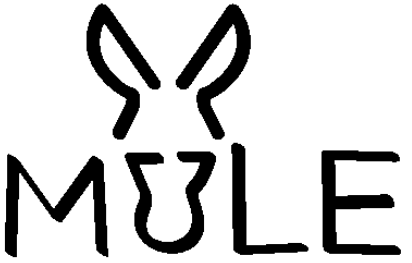
fix: set lifecyle data for stopped or failed nodes •••
René Zbinden authored 1 year ago

fix(agent): set LimitNOFILE=16384 in systemd
René Zbinden authored 1 year ago

fix: only log action for current node •••
René Zbinden authored 1 year ago

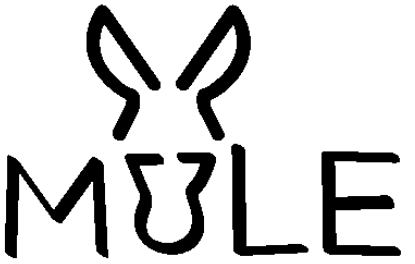
```
{"level":"INFO","ts":"2023-05-02T13:29:18.349+0200","caller":"repo/repo.go:177","msg":"put object","key":"ins
inition":"systemd","name":"chronyd.service","current-status":"running","target-status":"running","override-st
panic: assignment to entry in nil map
goroutine 259 [running]:
gitlab.pnet.ch/mule/mule/internal/metrics.ProfilesCollector.collectEmptyProfiles({0xc000012390, 0xc000480f80,
        /appl/sdsa/gitlab-runner/builds/eqWDYyC6/0/mule/mule/internal/metrics/profiles.go:279 +0x4ed
gitlab.pnet.ch/mule/mule/internal/metrics.ProfilesCollector.Start({0xc000012390, 0xc000480f80, 0xc0001246f0,
        /appl/sdsa/gitlab-runner/builds/eqWDYyC6/0/mule/mule/internal/metrics/profiles.go:96 +0x15e
created by gitlab.pnet.ch/mule/mule/internal/http/api.(*API).startMetricsCollector
        /appl/sdsa/gitlab-runner/builds/eqWDYyC6/0/mule/mule/internal/http/api/api.go:491 +0x1ca
```

One Big Bug



- June 2022 (after serveral weeks in production)
- Incident: The GUI is unresponsive
- Server stopped handling Jetstream messages
- No Errors in Logs
- OS Patch of NATS Servers

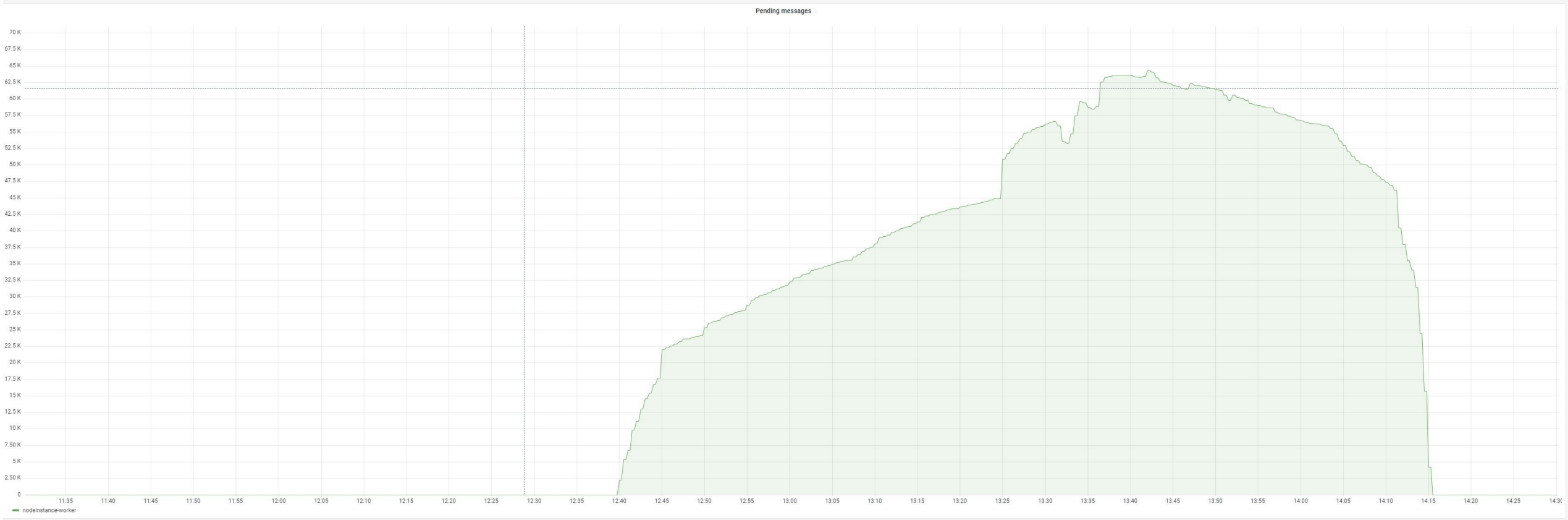
Select isn’t Broken



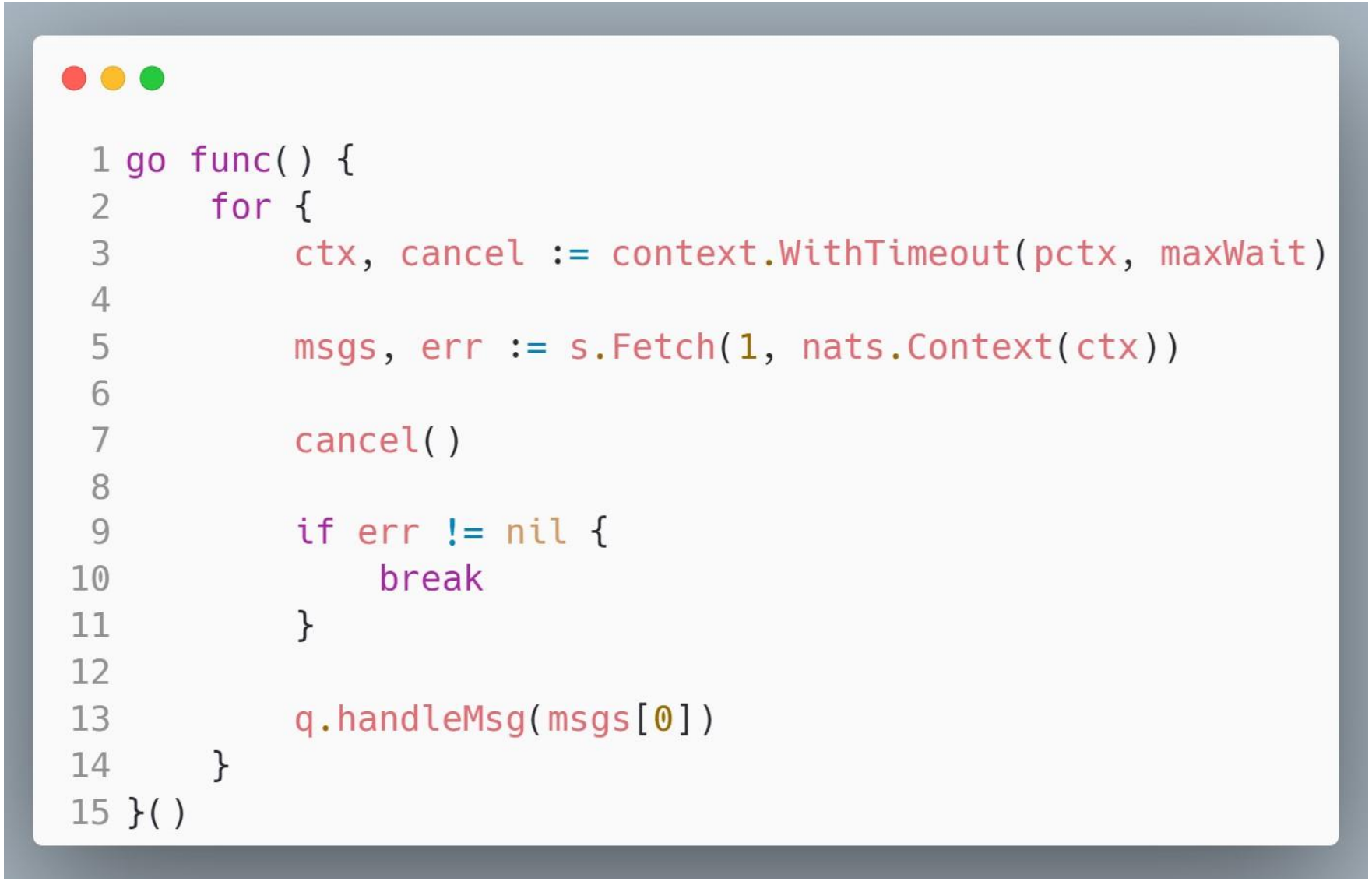
Programmers will sometimes think the problems they encounter are the fault of bugs in well-tested system software, rather than their own code.

The Pragmatic Programmer, by Hunt and Thomas

Pending Messages



Bug

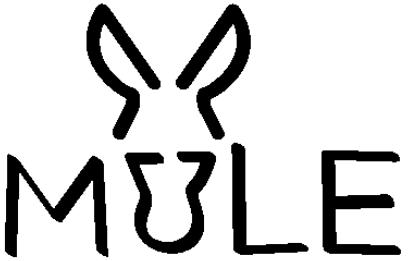


```go
go func() {
    for {
        ctx, cancel := context.WithTimeout(pctx, maxWait)

        msgs, err := s.Fetch(1, nats.Context(ctx))

        cancel()

        if err != nil {
            break
        }

        q.handleMsg(msgs[0])
    }
}()
```

Bug Fix

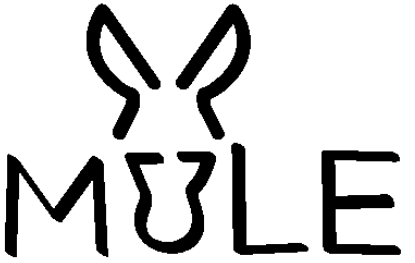
```go
go func() {
    for {
        ctx, cancel := context.WithTimeout(pctx, maxWait)

        msgs, err := s.Fetch(1, nats.Context(ctx))

        cancel()

        if err != nil {
            l.Errorw("failed to get next message", "err", err)

            continue
        }

        q.handleMsg(msgs[0])
    }
}()
```

Refactorings in the Future



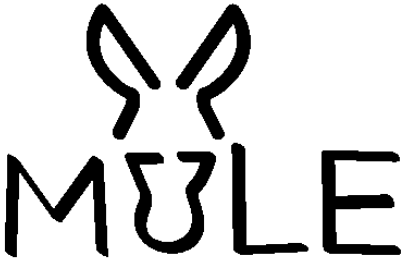
- Replace `uber-go/zap` logger with `log/slog` from stdlib
- Get rid of `google.golang.org/grpc` dependency
  - Replace with `connectrpc/connect-go`
    - Built on `net/http`
    - Automatic protocol detection
    - Semantic versioning
    - `grpc-web` support
- Tracing?

Conclusion



- Go was the right choice.
  - Static type system (refactoring)
  - Easy cross compilation (Windows and even Solaris)
  - Static binaries for easy deployment
- NATS was the right choice.
  - Synthetic load test with ~10k clients
- It’s hard to predict the load of a new distributed system, with unknown load requirements.

Why NATS?



- Simplicity (easy to deploy and monitor)
- High Performance and low latency
- Scalability and resilience (scale horizontally, built-in fault tolerance)
- Many communication patterns (request-reply)
- Testability

```
func TestWithNats(t *testing.T) {
    start := time.Now()
    _ = natstest.RunRandClientPortServer()
    fmt.Println(time.Since(start))
}
```

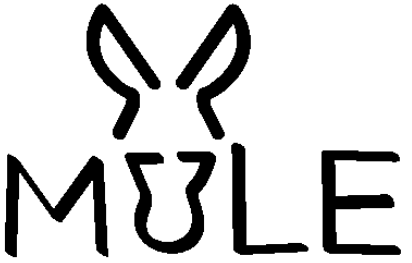
Minor Issues with NATS

Server Overview

Host	Version	JS	Conns	Subs	Routes	GWs	Mem	CPU %	Cores	Slow	Uptime	RTT
[redacted]	2.9.17	yes	1,343	8,195	2	0	323 MiB	3	16	1	70d23h29m12s	1ms
[redacted]	2.9.17	yes	1,183	8,037	2	0	283 MiB	3	16	1	70d22h28m14s	3ms
[redacted]	2.9.17	yes	819	7,678	2	0	213 MiB	2	16	1	52d4h51m29s	3ms
3		3	3,345	23,910			819 MiB			3		

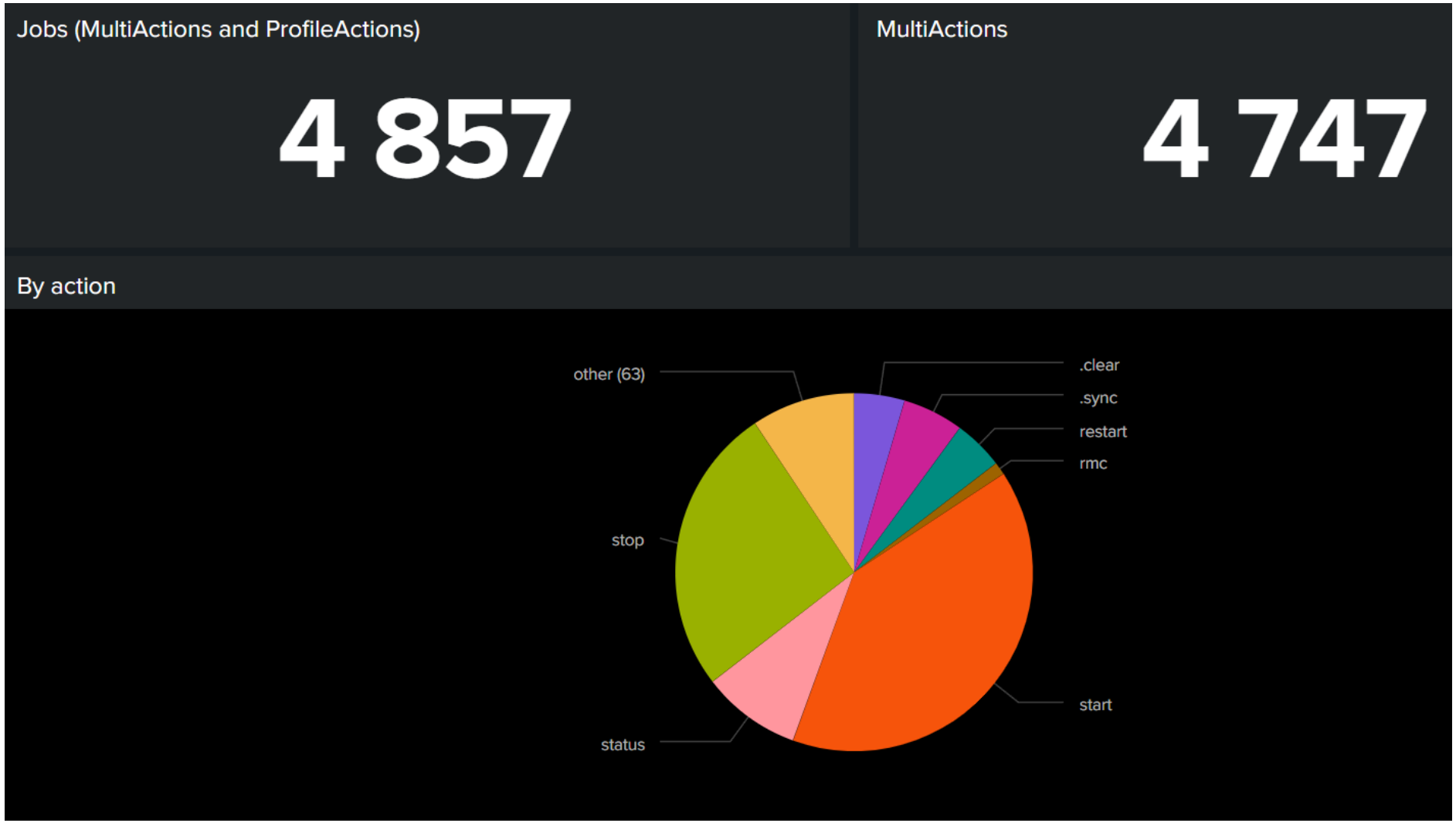
Fixed with 2.10.x

Refactoring - NATS



- Replaces the communication between backend and agent
- Etcd is the only holder of the runtime state for the GUI
- JetStream for state changes
- NATS for the action queue and configuration requests

Statistics


Jobs (MultiActions and ProfileActions)
4 857
MultiActions
4 747
By action
other (63)
.clear
.sync
restart
rmc
stop
start
status

NATS - Architecture