# Pandora Documentation

**Alexey Lavrenuke, Vladimir Skipor**

**Aug 30, 2018**

# Contents:


**1 Installation** **3**

**2 Your first test** **5**
2.1 Config file . . . . . 5
2.2 References . . . . . 6

**3 Custom guns** **7**

**4 Pandora's performance** **13**
4.1 HTTP requests to nginx . . . . . 13
4.2 Custom scenarios . . . . . 16

**5 Architectural overview** **19**
5.1 Architectural scheme . . . . . 19
5.2 Component types . . . . . 20

Pandora is a high-performance load generator in Go language. It has built-in HTTP(S) and HTTP/2 support and you can write your own load scenarios in Go, compiling them just before your test.

# CHAPTER 1

# Installation

Download binary release or build from source.

We use dep for package management. Install it before proceeding. Then build a binary with go tool (use go >= 1.8.3):

```
go get github.com/yandex/pandora
cd $GOPATH/src/github.com/yandex/pandora
dep ensure
go install
```

You can also cross-compile for other arch/os:

```
GOOS=linux GOARCH=amd64 go build
```

# CHAPTER 2

# Your first test

You can use Pandora alone or use it with Yandex.Tank[2] as a test runner and Overload[1] as a result viewer. In the second case Pandora's configuration is the same, but you will embed it into Yandex.Tank's config.

## 2.1 Config file

Pandora supports config files in YAML[3] format. Create a new file named `load.yaml` and add following lines in your favourite editor:

```
pools:
  - id: HTTP pool                    # pool name (for your choice)
    gun:
      type: http                     # gun type
      target: example.com:80         # gun target
    ammo:
      type: uri                      # ammo format
      file: ./ammo.uri               # ammo File
    result:
      type: phout                    # report format (phout is compatible with Yandex.
↪Tank)
      destination: ./phout.log       # report file name

    rps:                             # shooting schedule
      type: line                     # linear growth
      from: 1                        # from 1 response per second
      to: 5                          # to 5 responses per second
      duration: 60s                  # for 60 seconds

    startup:                         # instances startup schedule
```

(continues on next page)

[2] http://yandextank.readthedocs.org/en/latest/configuration.html#pandora
[1] https://overload.yandex.net
[3] https://en.wikipedia.org/wiki/YAML

(continued from previous page)

```
    type: once                      # start 10 instances
    times: 10
```

*ammo.uri*:

```
/my/first/url
/my/second/url
```

Run your tests:

```
pandora load.yaml
```

The results are in `phout.log`. Use Yandex.Tank[2] and Overload[1] to plot them.

## 2.2 References

# CHAPTER 3

## Custom guns

You can create you own Golang-based gun with *pandora*.

There is an example of custom gun shooting via gRPC.

We create a new gun and define `shoot` method for it w/ our test logic.

```
// create a package
package main

// import some pandora stuff
// and stuff you need for your scenario
// and protobuf contracts for your grpc service

import (
        "github.com/golang/protobuf/ptypes/timestamp"
        "github.com/satori/go.uuid"
        "github.com/spf13/afero"
        "github.com/yandex/pandora/cli"
        "github.com/yandex/pandora/components/phttp/import"
        "github.com/yandex/pandora/core"
        "github.com/yandex/pandora/core/aggregator/netsample"
        "github.com/yandex/pandora/core/import"
        "github.com/yandex/pandora/core/register"
        "google.golang.org/grpc"
        "log"
        "context"
        "strconv"
        "strings"
        "time"
        pb "my_package/my_contracts"
)

type Ammo struct {
        Tag         string
        Param1      string
```

(continues on next page)

(continued from previous page)

```
        Param2      string
        Param3      string
}

type Sample struct {
        URL              string
        ShootTimeSeconds float64
}

type GunConfig struct {
        Target string `validate:"required"` // Configuration will fail, without␣
↪target defined
}

type Gun struct {
        // Configured on construction.
        client grpc.ClientConn
        conf   GunConfig
        // Configured on Bind, before shooting
        aggr core.Aggregator // May be your custom Aggregator.
        core.GunDeps
}

func NewGun(conf GunConfig) *Gun {
        return &Gun{conf: conf}
}

func (g *Gun) Bind(aggr core.Aggregator, deps core.GunDeps) error {
        // create gRPC stub at gun initialization
        conn, err := grpc.Dial(
                g.conf.Target,
                grpc.WithInsecure(),
                grpc.WithTimeout(time.Second),
                grpc.WithUserAgent("load test, pandora custom shooter"))
        if err != nil {
                log.Fatalf("FATAL: %s", err)
        }
        g.client = *conn
        g.aggr = aggr
        g.GunDeps = deps
        return nil
}

func (g *Gun) Shoot(ammo core.Ammo) {
        customAmmo := ammo.(*Ammo)
        g.shoot(customAmmo)
}


func (g *Gun) case1_method(client pb.MyClient, ammo *Ammo) int {
        code := 0
        // prepare list of ids from ammo
        var itemIDs []int64
        for _, id := range strings.Split(ammo.Param1, ",") {
                if id == "" {
                        continue
                }
```

(continues on next page)

(continued from previous page)

```
                itemID, err := strconv.ParseInt(id, 10, 64)
                if err != nil {
                        log.Printf("Ammo parse FATAL: %s", err)
                        code = 314
                }
                itemIDs = append(itemIDs, itemID)
        }

        out, err := client.GetSomeData(
                context.TODO(), &pb.ItemsRequest{
                        itemIDs})

        if err != nil {
                log.Printf("FATAL: %s", err)
                code = 500
        }

        if out != nil {
                code = 200
        }
        return code
}

func (g *Gun) case2_method(client pb.MyClient, ammo *Ammo) int {
        code := 0
        // prepare item_id and warehouse_id
        item_id, err := strconv.ParseInt(ammo.Param1, 10, 0)
        if err != nil {
                log.Printf("Failed to parse ammo FATAL", err)
                code = 314
        }
        warehouse_id, err2 := strconv.ParseInt(ammo.Param2, 10, 0)
        if err2 != nil {
                log.Printf("Failed to parse ammo FATAL", err2)
                code = 314
        }

        items := []*pb.SomeItem{}
        items = append(items, &pb.SomeItem{
                item_id,
                warehouse_id,
                1,
                &timestamp.Timestamp{time.Now().Unix(), 111}
        })

        out2, err3 := client.GetSomeDataSecond(
                context.TODO(), &pb.SomeRequest{
                        uuid.Must(uuid.NewV4()).String(),
                        1,
                        items})
        if err3 != nil {
                log.Printf("FATAL", err3)
                code = 316
        }

        if out2 != nil {
                code = 200
```

(continues on next page)

(continued from previous page)

```
        }


        return code
}

func (g *Gun) shoot(ammo *Ammo) {
        code := 0
        sample := netsample.Acquire(ammo.Tag)

        conn := g.client
        client := pb.NewClient(&conn)

        switch ammo.Tag {
        case "/MyCase1":
            code = g.case1_method(client, ammo)
        case "/MyCase2":
            code = g.case2_method(client, ammo)
        default:
            code = 404
        }

        defer func() {
                sample.SetProtoCode(code)
                g.aggr.Report(sample)
        }()
}

func main() {
        //debug.SetGCPercent(-1)
        // Standard imports.
        fs := afero.NewOsFs()
        coreimport.Import(fs)
        // May not be imported, if you don't need http guns and etc.
        phttp.Import(fs)

        // Custom imports. Integrate your custom types into configuration system.
        coreimport.RegisterCustomJSONProvider("custom_provider", func() core.Ammo {␣
↪return &Ammo{} })

        register.Gun("My_custom_gun_name", NewGun, func() GunConfig {
                return GunConfig{
                        Target: "default target",
                }
        })

        cli.Run()
}
```

Now it's time to compile our gun. Install deps and compile your custom gun file (`go build my_custom_gun.go`). After that step you'll get `my_custom_gun` binary file, it is compiled pandora with your custom gun inside.

Now its time to create `load.yaml`:

```
pools:
    - id: HTTP pool
        gun:
```

(continues on next page)

(continued from previous page)

```
        type: My_custom_gun_name   # custom gun name specified
        target: "your_grpc_host:your_grpc_port"
    ammo:
        type: custom_provider
        source:
            type: file
            path: ./json.ammo
    result:
        type: phout
        destination: ./phout.log
    rps: {duration: 30s, type: line,  from: 1, to: 2}
    startup:
        type: once
        times: 10
    log:
        level: error
```

And create ammofile `./json.ammo`:

```
{"tag": "/MyCase1", "Param1": "146837693,146837692,146837691"}
{"tag": "/MyCase2", "Param2": "555", "Param1": "500002"}
```

We are ready to shoot. Try it.

# CHAPTER 4

# Pandora's performance

Alexander Ivanov made some performance tests for the gun itself. Here are the results.

- Server: NGinx, 32 cores, 64G RAM.
- Tank: 32 cores, 128G RAM.
- Network: 1G.

## 4.1 HTTP requests to nginx

Static pages with different sizes. Server delays implemented in Lua script, we can set delay time using `sleep` query parameter:

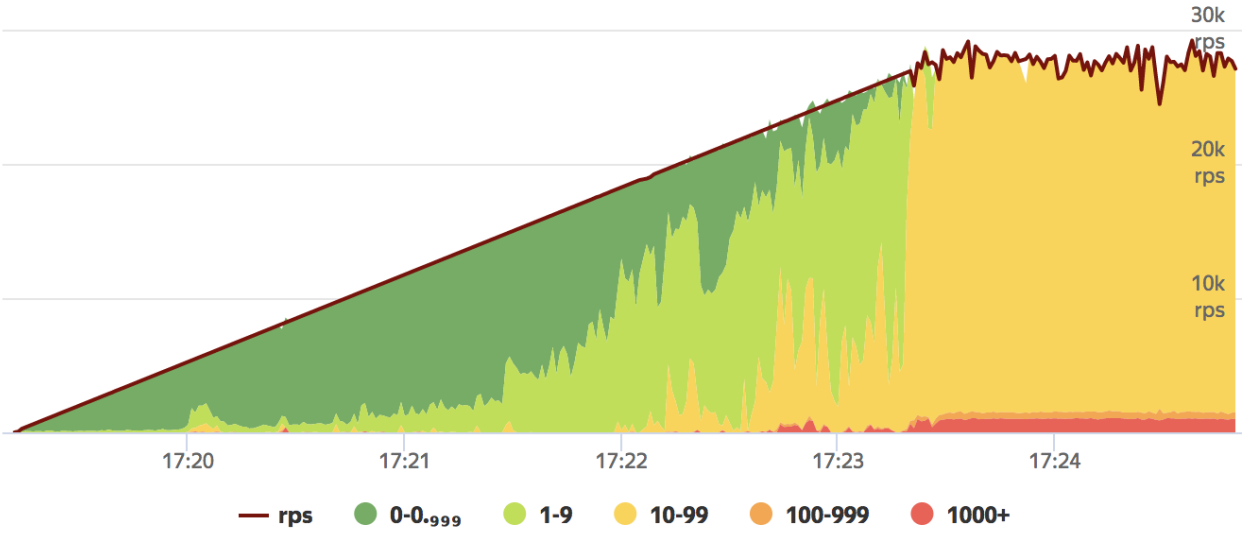
```
server {
    listen          12999      default;
    listen          [::]:12999 default         ipv6only=on;
    server_name     pandora.test.yandex.net;

    location ~* / {

        rewrite_by_lua_block {
            local args = ngx.req.get_uri_args()
            if args['sleep'] then
                        ngx.sleep(args['sleep']/1000)
            end;
        }

        root /etc/nginx/pandora;
        error_page 404 = 404;

        }

        access_log off;
```

(continues on next page)

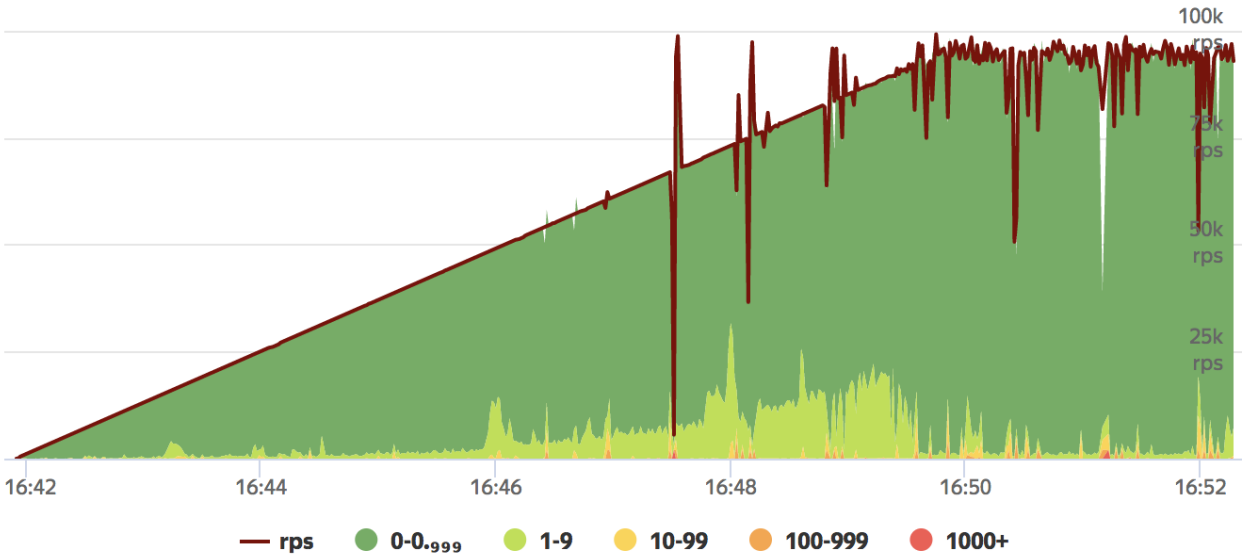
(continued from previous page)

```
        error_log off;
}
```

- **Connection: Close** 23k RPS

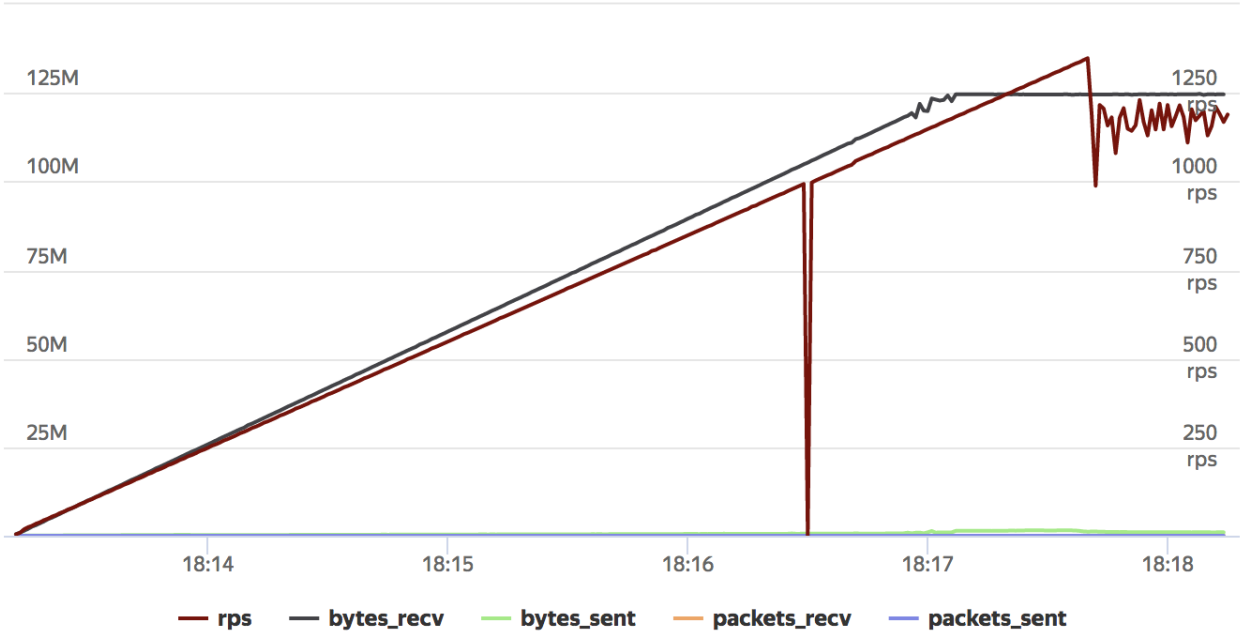



- **Connection: Keep-Alive** 95k RPS



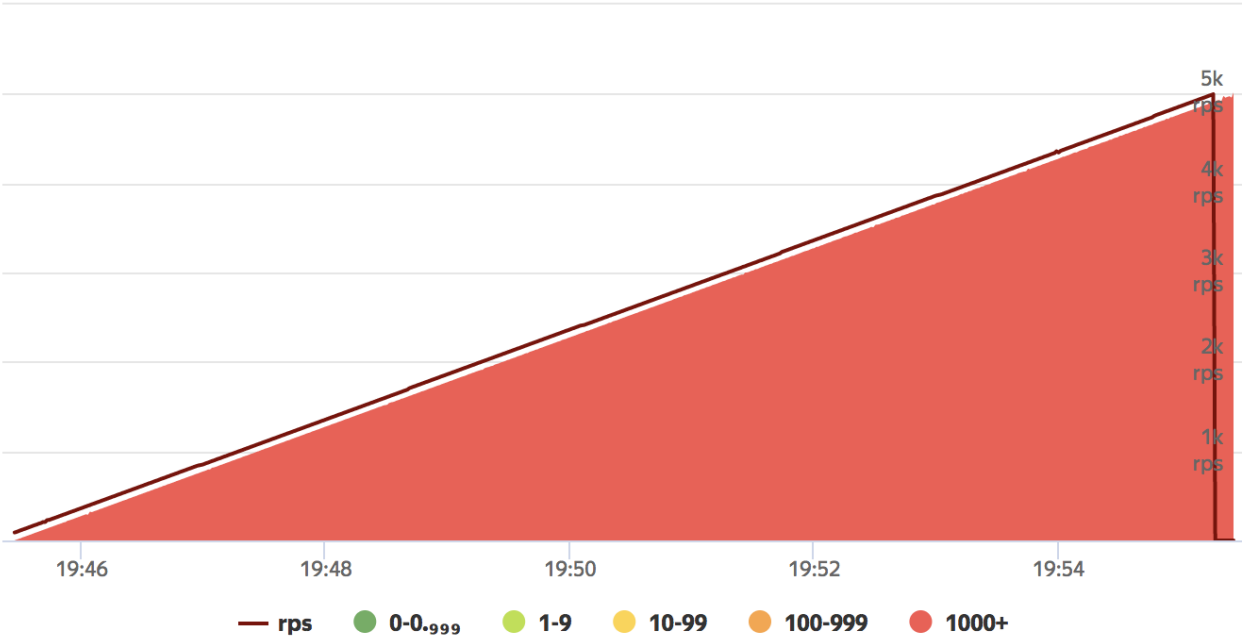

- **Response size 10kB** maxed out network interface. OK.
- **Response size 100kb** maxed out network interface. OK.
- **POST requests 10kB** maxed out network interface. OK.

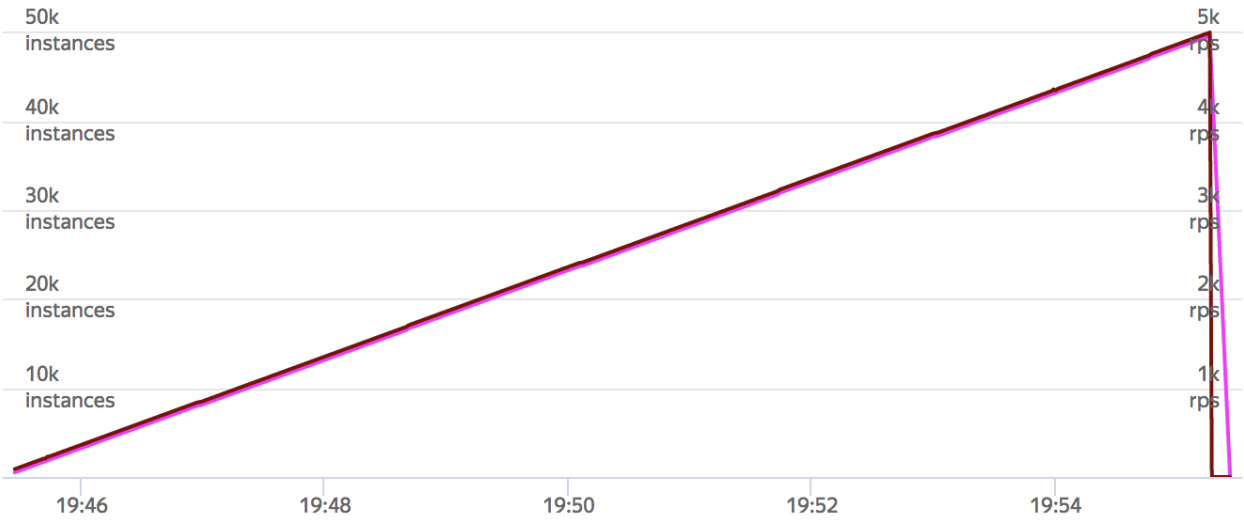
- **POST requests 100kB** maxed out network interface. OK.
- **POST requests 1MB** maxed out network interface. OK.



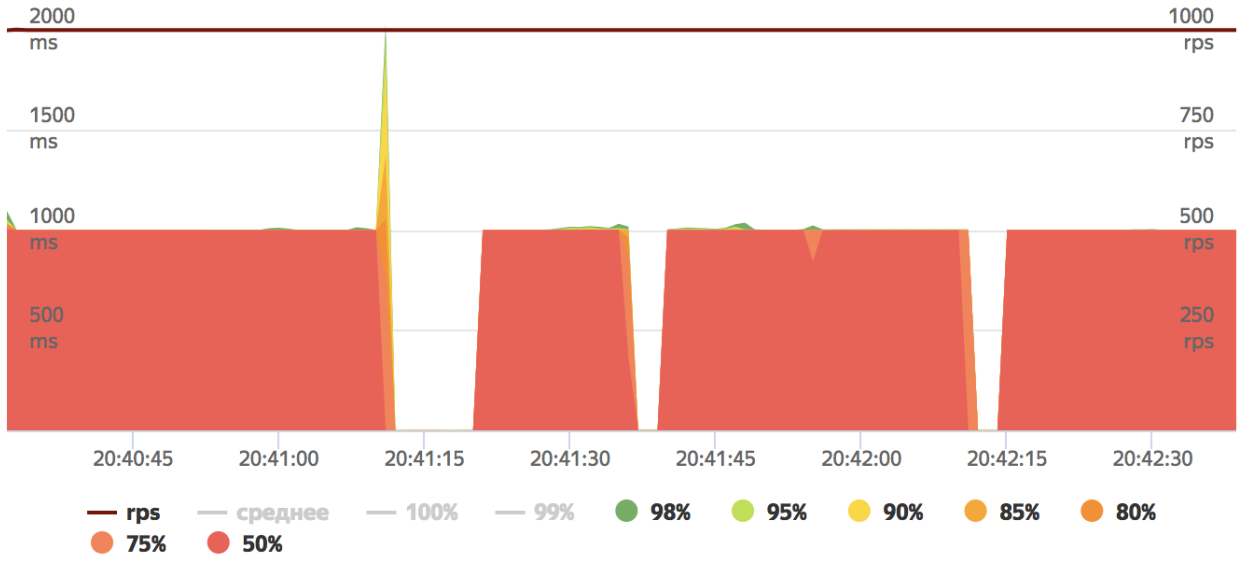

- **50ms server delay** 30k RPS. OK.
- **500ms server delay** 30k RPS, 30k instances. OK.
- **1s server delay** 50k RPS, 50k instances. OK.
- **10s server delay** 5k RPS, 5k instances. OK.

**All good.**






- **Server fail during test** OK.

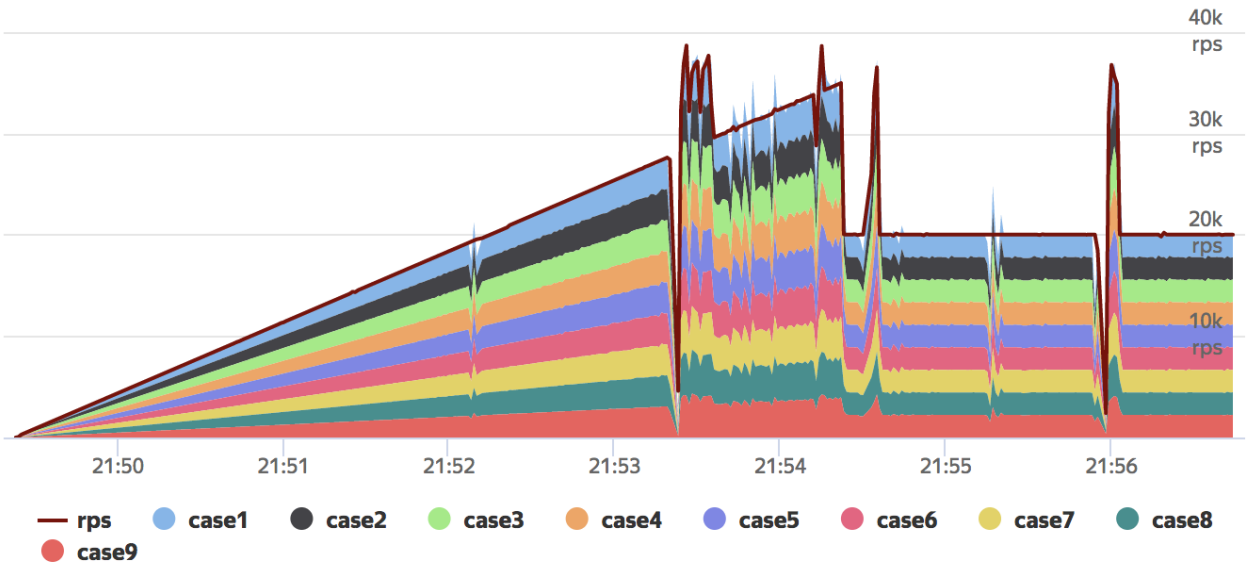



## 4.2 Custom scenarios

Custom scenarios performance depends very much of their implementation. In some our test we saw spikes caused by GC. They can be avoided by reducing allocation size. It is a good idea to optimize your scenarios. Go has a lot of tools helping you to do this.

**Note:** We used JSON-formatted ammo to specify parameters for each scenario run.

- **Small requests** 35k RPS. OK.
- **Some scenario steps with big JSON bodies** 35k RPS. OK.

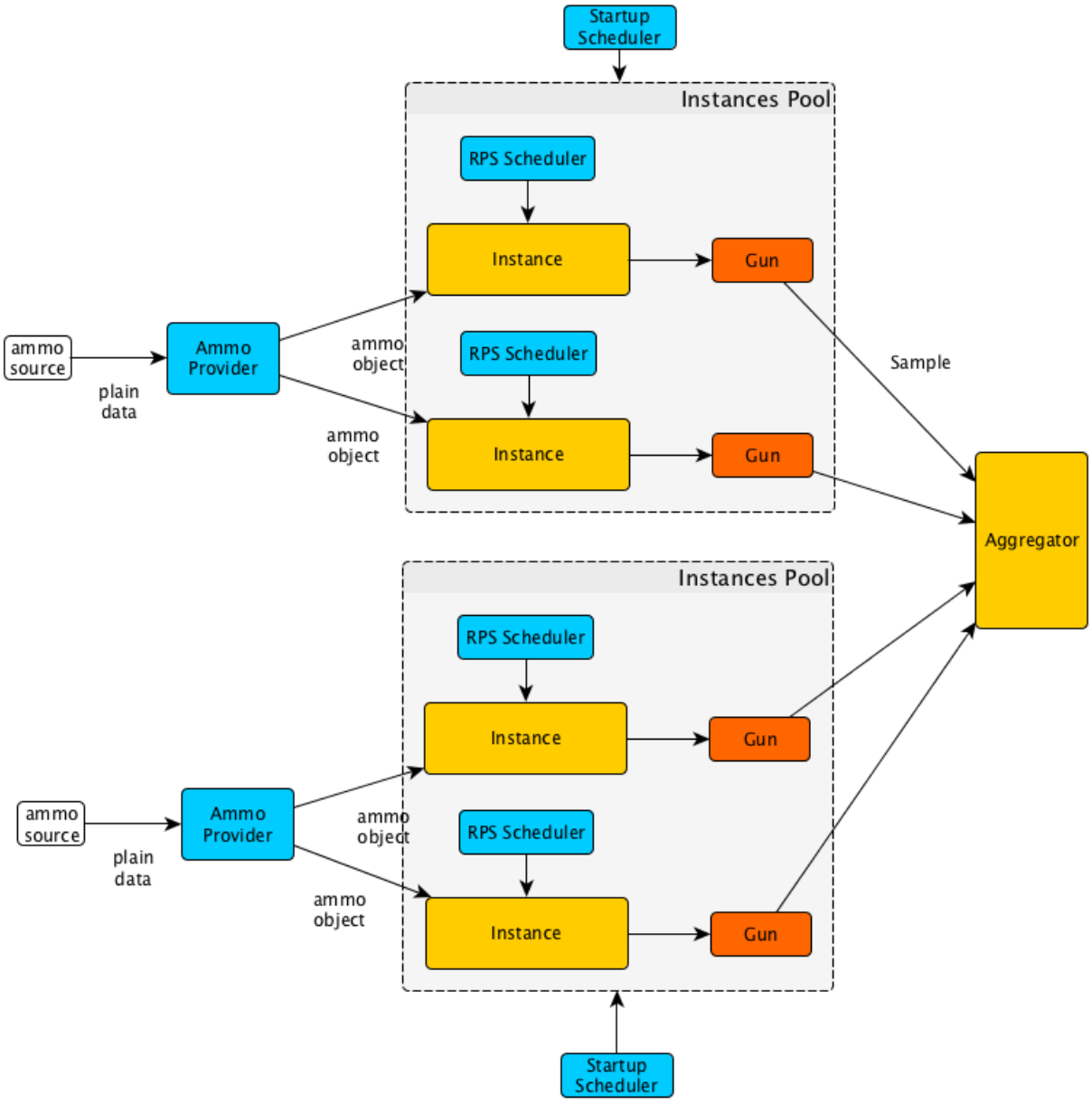


CHAPTER 5

# Architectural overview

## 5.1 Architectural scheme

You can download architectural scheme source here. Open it with YeD editor.




Pandora is a set of components talking to each other through Go channels. There are different types of components.

## 5.2 Component types

### 5.2.1 Ammo Provider

Ammo Provider knows how to make an ammo object from an ammo file or other external resource. Instances get ammo objects from Ammo Provider.

### 5.2.2 Instances Pool

Instances Pool manages the creation of Instances. You can think of one Instance as a single user that sends requests to a server sequentially. All Instances from one Instances Pool get their ammo from one Ammo Provider. Instances creation times are controlled by Startup Scheduler. All Instances from one Instances Pool also have Guns of the same type.

### 5.2.3 Scheduler

Scheduler controls other events' times by pushing messages to its underlying channel according to the Schedule. It can control Instances startup times, RPS amount (requests per second) or other processes.

By combining two types of Schedulers, RPS Scheduler and Instance Startup Scheduler, you can simulate different types of load. Instace Startup Scheduler controls the level of parallelism and RPS Scheduler controls throughput.

RPS Scheduler can limit throughput of a whole instances pool, i.e. 10 RPS on 10 instances means 10 RPS overall, or limit throughput of each instance in a pool individually, i.e. 10 RPS on each of 10 instances means 100 RPS overall.

If you set RPS Scheduler to 'unlimited' and then gradually raise the number of Instances in your system by using Instance Startup Scheduler, you'll be able to study the scalability of your service.

If you set Instances count to a big, unchanged value (you can estimate the needed amount by using Little's Law) and then gradually raise the RPS by using RPS Scheduler, you'll be able to simulate Internet and push your service to its limits.

You can also combine two methods mentioned above.

### 5.2.4 Instances and Guns

Instances takes an ammo, waits for a Scheduler tick and then shoots with a Gun it has. Gun is a tool that sends a request to your service and measures the parameters (time, error codes, etc.) of the response.

### 5.2.5 Aggregator

Aggregator collects measured samples and saves them somewhere.