Maestro Documentation

Release 9.6.1

Top Free Games

Sep 29, 2021

Contents

CHAPTER 1

Overview

Maestro is a Kubernetes-native game server scheduler.

1.1 Goal

Maestro goal is to provide an unified system that automatically scales game rooms, regardless of the transport layer protocol (TCP, UDP).

This system is related to a matchmaker but does not handle the specificities of a match such as how many players fit in a room. It only deals with high level room occupation, i.e. is the room occupied or available. The rooms communicate directly with the matchmaker in order to register and unregister themselves from the matchmaking.

1.2 Ecosystem

The Maestro ecosystem is composed by:

- maestro: the service itself
- maestro-cli: a wrapper for the maestro-api endpoints
- maestro-client: a client lib for Unity and cocos2dx, responsible for calling maestro HTTP routes defined in the *room protocol*. It also must catch sigterm/sigkill and handle the room graceful shutdown.

In the future, we may have an UI for displaying metrics such as percentage of rooms usage, room occupation rates and rooms resource metrics, like CPU and memory.

1.3 Definitions

Maestro uses some abstractions, based on Kubernetes resources, in order to implement its service.

A **Game Room Unity (GRU)** is where the game server logic will run and clients will connect to execute their matches. It's the most atomic entity in a Maestro architecture.

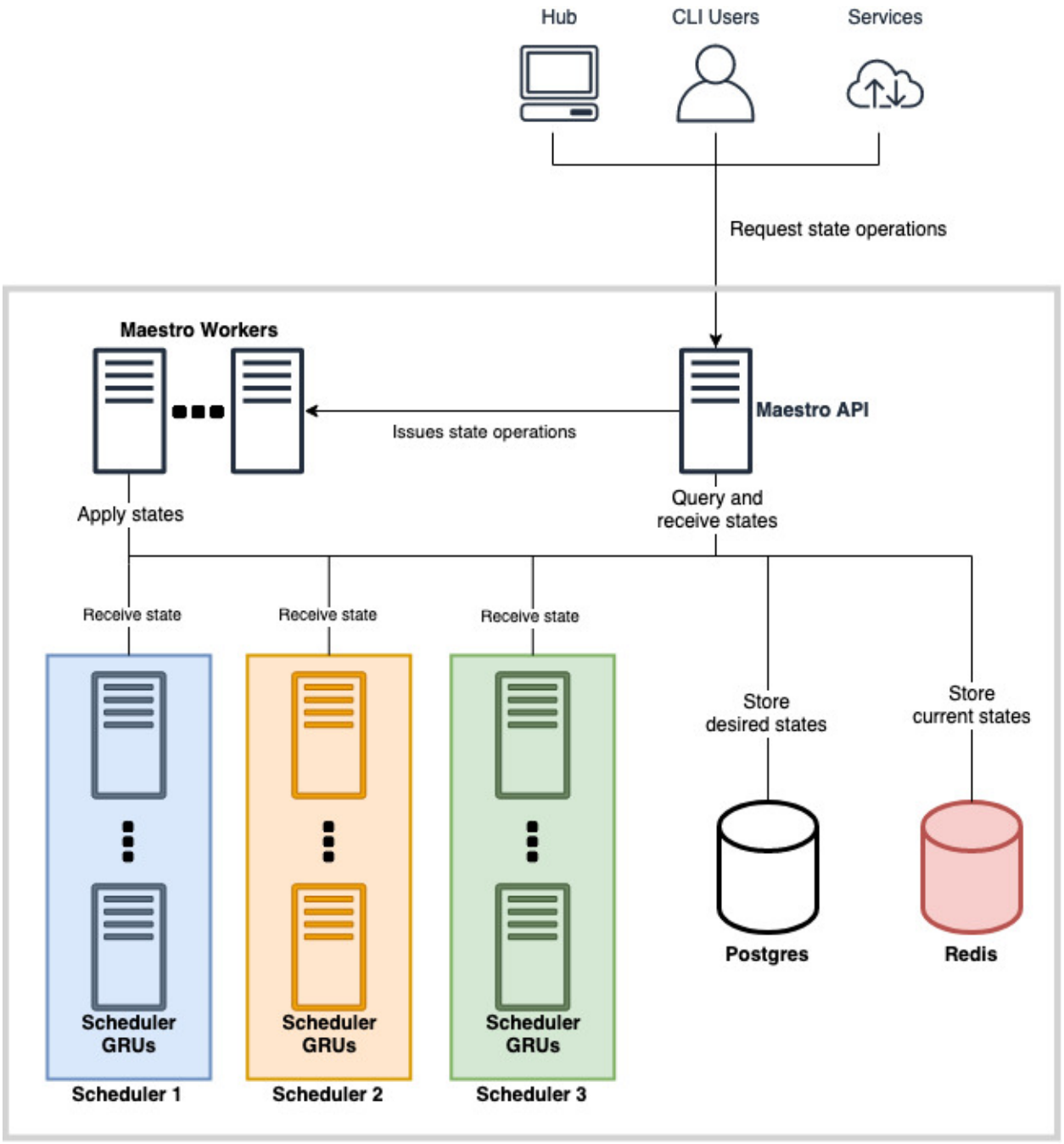
One could define a GRU as a Kubernetes service (type nodePort) associated with a single pod. This restriction is made because in AWS we cannot load balance UDP. We're using containerized applications and Kubernetes in order to simplify the management and scaling of the game rooms.

A **scheduler** is a specification for a set of GRUs. It defines the image, variables, computing and networks resources, permissions and other configurations for these GRUs. One can define them as a Kubernetes deployment strongly attached to a namespace. This restriction is made because Maestro does not allow more than one schedulers to be defined in the same namespace.

1.4 Architecture

Here, we provide an overview of the service architecture. Maestro requires that these entities, except Postgres and Redis, run in the same Kubernetes cluster.

The diagram below depicts the integration between services and we provide a description of each component responsibility.



diagram

1.4.1 API

The API is the connection of Maestro to the external world and with the GRUs themselves. It talks HTTP and is responsible for:

- Receiving and processing client operation requests over schedulers;
- Listening to *GRU status*, through healthchecks;

- Storing scheduler desired states in Postgres.

1.4.2 Postgres

Postgres is the storage for schedulers' desired state. They hold the actual configuration a scheduler should have for their GRUs, keeping track of the progress status of this configurations, its version and for what game they should be applied. At the end, GRUs of a given scheduler should reflect the state described at Postgres.

1.4.3 Redis

Redis is responsible for caching the Kubernetes cluster state, obtained through Kubernetes API. Hence, it reflects Maestro's schedulers current states, preventing Maestro API and workers from flooding Kubernetes API.

Maestro API and workers would frequently consult this storage in order to obtain schedulers state and checking if they are already matching the desired state at Postgres. If they don't, they would perform operations in order to reach these states and update Redis according.

1.4.4 Worker

Workers implement the core logic of Maestro, being responsible of guaranteeing that old states are deleted, new states are applied and scaling policies are being followed.

As soon it starts, the worker obtain schedulers' states from Postgres and start a **watcher process** for each scheduler. Through a loop, a worker will guarantee that each scheduler will its own watcher process, creating new process for new schedulers and keeping already created ones healthy.

Since workers are horizontally scaled, its possible a scheduler has more than one watcher process managing it, across different workers. Given that, Maestro implement locks, which are responsible for avoiding race conditions between scheduler watcher process along distinct workers instances.

Watcher Process

The watcher process is instanced at each worker and is responsible for managing a single scheduler. Through loops, it executes the following responsibilities:

1. **Update the scheduler GRU states at Redis.** It does that querying Maestro API, in a separate routine of its own;
2. **Guarantee that GRUs states are in sync with Postgres.** It does using the scheduler configuration version. Whenever GRUs are not using the desired version, the watcher executes routines to ensure that. Major versions changes trigger pods updates, while minor and patch version doesn't;
3. **Remove dead GRUs.** It does that by checking which GRUs are not signaling a healthy state between the configured healthcheck interval. Dead rooms are commanded to die and their data are cleaned from Redis.
4. **Execute auto scaling policies.** It does that by checking used resources and rooms over scheduler GRUs and deciding if upscales or downscales should be applied.

1.5 Configuring Maestro

The maestro binary receives a list of config files and spawn one maestro-controller for each config.

The config file must have the following information:

- Docker image
- Autoscaling policies
- Manifest yaml template
    1. Default configuration (ENV VARS)
    2. Ports and protocols (UDP, TCP)
    3. Resources requests (cpu and memory)

Example yaml config:

```
name: pong-free-for-all      # this will be the name of the kubernetes namespace (it␣
↪must be unique)
game: pong                   # several configs can refer to the same game
image: pong/pong:v123
affinity: node-affinity      # optional field: if set, rooms will be allocated␣
↪preferentially to nodes with label "node-affinity": "true"
toleration: node-toleration # optional field: if set, rooms will also be allocated in␣
↪nodes with this taint
occupiedTimeout: match-time # how much time a match has. If room stays with occupied␣
↪status for longer than occupiedTimeout seconds, the room is deleted
ports:
  - containerPort: 5050      # port exposed in the container
    protocol: UDP            # supported protocols are TCP and UDP
    name: gamebinary         # name identifying the port (must be unique for a config)
  - containerPort: 8888
    protocol: TCP
    name: websocket
requests:                    # these will be the resources requests applied to the␣
↪pods created in kubernetes
  memory: 1Gi                # they are used to calculate resource(cpu and memory)␣
↪usage and trigger autoscaling when metrics triggers are defined
  cpu: 1000m
limits:                      # these will be the resources limits applied to the pods␣
↪created in kubernetes
  memory: "128Mi"            # they are used to decide how many rooms can run in each␣
↪node
  cpu: "1"                   # more info: https://kubernetes.io/docs/tasks/configure-
↪pod-container/assign-cpu-ram-container/
shutdownTimeout: 180         # duration in seconds the pod needs to terminate␣
↪gracefully
autoscaling:
  min: 100                   # minimum amount of GRUs
  max: 1000                  # maximum amount of GRUs
  up:
    metricsTrigger:          # list of triggers that define the autoscaling behaviour
      - type: room           # the triggers can be of type room, cpu or memory
        threshold: 80        # percentage of the points that are above 'usage' needed␣
↪to trigger scale up
        usage: 70            # minimum usage (percentage) that can trigger the scaling␣
↪policy
        time: 600            # duration in seconds to wait before scaling policy takes␣
↪place
    cooldown: 300            # duration in seconds to wait before consecutive scaling
  down:
    metricsTrigger:
      - type: cpu
```

(continues on next page)

(continued from previous page)

```
      threshold: 80       # percentage of the points that are above 'usage' needed␣
↪to trigger scale down
      usage: 50           # maximum usage (percentage) the can trigger the scaling␣
↪policy
      time: 900           # duration in seconds to wait before scaling policy takes␣
↪place
    cooldown: 300         # duration in seconds to wait before consecutive scaling
env:                      # environment variable to be passed on to the container
  - name: EXAMPLE_ENV_VAR
    value: examplevalue
  - name: ANOTHER_ENV_VAR
    value: anothervalue
cmd:                      # if the image can run with different arguments you can␣
↪specify a cmd
  - "./room-binary"
  - "-serverType"
  - "6a8e136b-2dc1-417e-bbe8-0f0a2d2df431"
forwarders:               # optional field: if set events will be forwarded for the␣
↪grpc matchmaking plugin
  grpc:
    matchmaking:
      enabled: true
      metadata:           # the forwarder metadata is forwarded in scheduler events␣
↪(create and update)
        matchmakingScript: default
        metadata:
          authTimeout: 10000
        minimumNumberOfPlayers: 1
        numberOfTeams: 1
        playersPerTeam: 6
        roomType: "10"
        tags:
          score: score
```

A JSON file equivalent to the yaml above can also be used.

1.6 Room Protocol

Game rooms have four different statuses:

- **Creating**: from the time maestro starts creating the GRU in Kubernetes until a room ready is received.
- **Ready**: from the time room ready is called until a match started is received. It means the room is available for matches.
- **Occupied**: from the time match started is called until a match ended is received. It means the room is not available for matches.
- **Terminating**: from the time a sigkill/sigterm signal is received by the room until the GRU is no longer available in Kubernetes.

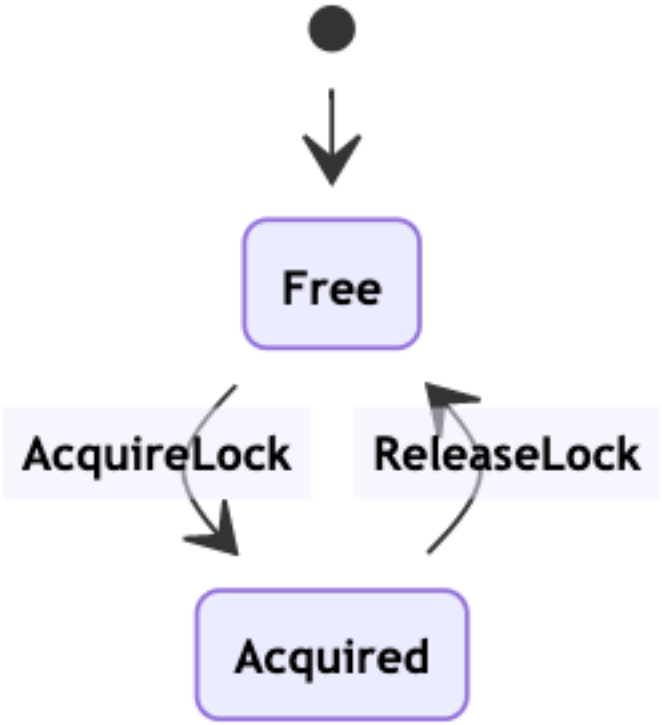
Maestro's auto scaling policies are based on the number of rooms that are in ready state.

CHAPTER 2

Locks

As described in the *Overview* page, Maestro schedulers may have multiple watcher process taking care of them. Hence, when an operation is meant to take place in a scheduler, these process can overlap and race conditions may appear. Given that, Maestro implements locks, used by some operations to prevent these conditions.

An operation that has locks usually checks to see if the lock is free: if not, the operation does not proceed. When the lock is free, the first thing the operation does it to acquire the lock, leaving it no longer free. In the end, the operation is also responsible for releasing the lock, usually when everything is done (including in error situations). The diagram below shows the simple flow it follows.



diagram

Those locks are stored at Redis and exposed through `/scheduler/{schedulerName}/locks` endpoint. The only exception is the panic lock, which is not present on this API.

2.1 Scheduler Config Lock

This lock is used to perform updates on scheduler configuration. All updates acquire this lock, but they might be released at different moment:

1. If changes are related to the pod itself (like image and port changes), Maestro starts a replace on the pods to ensure that they're on the correct version and lock lasts until this replace operation is finished.
2. If changes don't affects the pods (like scaling configuration or permissions), the operation only consists of changing the scheduler in the database and the lock is released.

NOTE: Even if a scheduler update triggers a replace on the pods, this is done asynchronous by the API itself, so the update request returning doesn't mean that the lock should be already free.

Below, we present a pseudo-algorithm for understanding the lock flow. We name the Scale, Scheduler Config and Downscale locks as SL, CL and DL, respectively.

```
--- Acquire CL ---
  --- Acquire SL ---
    1 - load scheduler from database
    2 - check diff between new and old YAML
    3 - save new scheduler on database
  --- Release SL ---
  4 - if rolling update is not needed go to 9
  --- Acquire DL ---
    5 - divide current pods in chunks of size maxSurge (a percentage of current pods)
```

(continues on next page)

(continued from previous page)

```
    *** Start goroutines ***
      6 - for each pod in chunk create a new pod and delete an old one
    *** End goroutines ***
    7 - if all chunks finished go to 9
    8 - if rolling times out or is canceled or error (when a pod fails to be created),
↪ do a rollback (start the flow again with old config)
  --- Release DL ---
  9 - update scheduler veersion status (deployed, canceled, error, ...) on database
--- Release CL ---
10 - end
```

2.2 Downscaling Lock

This lock is for the scale down process and should happen only once at a time. This process can happen in three scenarios:

1. When the worker (watcher) performs a RemoveDeadRooms and checks for rooms that does not have a desired state. It then chooses to remove those rooms and acquire the lock. This lock is only released when all the pods marked to be delete are gone or a timeout happen.
2. When the worker (watcher) performs a AutoScale and checks the delta and decides that it will perform a down scale, and it acquire the lock. This lock is only released when all the pods marked to be delete are gone or a timeout happen. (it uses the ScaleDown function)
3. When a scheduler scale (operation) is performed and it results in a downscaling, at this moment the lock is acquired. It performs a similar flow as the previous scenario, so it wait until all the pods to be gone or a timeout to happen to release the lock. (it uses the ScaleDown function)

NOTE: The task of acquiring and releasing the lock is not made at ScaleDown function so actually in both scenarios mentioned this is performed in a different way (but the results is the same).

NOTE 2: During down scaling at worker (triggered at AutoScale) it also has the Termination (described later) lock. So on this case the process acquire and releases both locks.

2.3 Termination Lock

This locks was mainly thought to prevent any operation at pods to be performed while the scheduler is being deleted. But it also works like a scale up lock (when performed by the AutoScale). It happens in two cases:

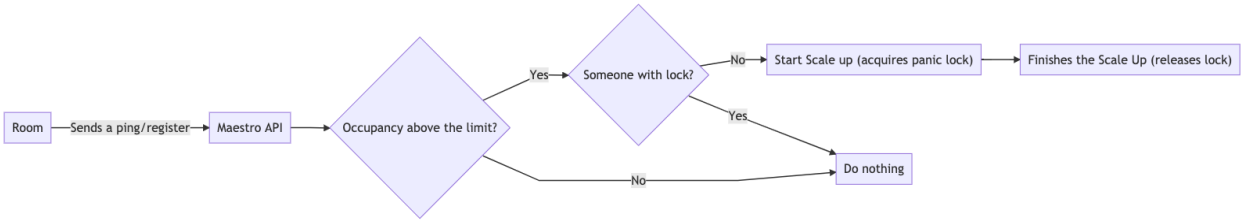
1. When a scheduler will be deleted, it acquires the lock before starting and release when the deletion is complete;
2. At the worker (watcher) the AutoScale process runs with this lock, so any up scale or down scale has this lock also. The lock is released when AutoScale finishes.

NOTE: The scheduler scale when performing an up scale does not acquire the termination lock, so it is possible to perform two up scales during the same time (even three if we count the panic scale).

2.4 Panic Lock

This process is used during a "desperate time" where Maestro is seeing the occupancy get above the limit defined by the scheduler, when this happens it does a "panic scale" using the upscale delta. This check happens every time a

rooms sends its status to Maestro (ping and register), so in order to prevent that multiple "panic scales" to happen it uses a lock. The "panic scale" flow is described as below:



diagram

CHAPTER 3

Hosting Maestro

3.1 Docker

Maestro needs to connect to a PostgreSQL database in order to persist schedulers configuration and state. The following environment variables must be specified:

- `MAESTRO_EXTENSIONS_PG_HOST` - PostgreSQL host to connect to;
- `MAESTRO_EXTENSIONS_PG_PORT` - PostgreSQL port to connect to;
- `MAESTRO_EXTENSIONS_PG_USER` - User of the PostgreSQL Server to connect to;
- `MAESTRO_EXTENSIONS_PG_PASS` - Password of the PostgreSQL Server to connect to;
- `MAESTRO_EXTENSIONS_PG_POOLSIZE` - PostgreSQL connection pool size;
- `MAESTRO_EXTENSIONS_PG_MAXRETRIES` - PostgreSQL connection max retries;
- `MAESTRO_EXTENSIONS_PG_DATABASE` - PostgreSQL database to connect to;
- `MAESTRO_EXTENSIONS_PG_CONNECTIONTIMEOUT` - Timeout for trying to establish connection;

Maestro also needs to connect to a Redis database in order to persist rooms statuses and lock watcher executions:

- `MAESTRO_EXTENSIONS_REDIS_URL` - Url of the Redis database to connect to;
- `MAESTRO_EXTENSIONS_REDIS_CONNECTIONTIMEOUT` - Timeout for trying to establish connection;

The watcher receives a few environment variables in order to configure itself:

- `MAESTRO_WATCHER_LOCKKEY` - String that will be as key for locking watcher executions;
- `MAESTRO_WATCHER_LOCKTIMEOUT` - Timeout for trying to acquire the watcher lock;
- `MAESTRO_WATCHER_GRACEFULSHUTDOWNTIMEOUT` - Timeout for graceful shutdown;
- `MAESTRO_WATCHER_AUTOSCALINGPERIOD` - Period (in seconds) for executing the watcher autoscaling;

The worker also receives an environment variable for configuration:

- `MAESTRO_WORKER_GRACEFULSHUTDOWNTIMEOUT` - Timeout for graceful shutdown;

- MAESTRO_WORKER_SYNCPERIOD - Period (in seconds) for executing the worker scheduler synchronization;

Other than that, there are a couple more configurations you can pass using environment variables:

- MAESTRO_SCALEUPTIMEOUTSECONDS - Timeout for trying to scale up a scheduler;
- MAESTRO_DELETETIMEOUTSECONDS - Timeout for trying to delete a scheduler;
- MAESTRO_PINGTIMEOUT - If a room sent the last PING request more than MAESTRO_PINGTIMEOUT seconds ago it is considered unresponsive and removed from the scheduler;

If you wish Sentry integration simply set the following environment variable:

- MAESTRO_SENTRY_URL - Sentry Client Key (DSN);

If you wish NewRelic integration you must set the following environment variables:

- MAESTRO_NEWRELIC_APP - NewRelic app name;
- MAESTRO_NEWRELIC_KEY - NewRelic key;

CHAPTER 4

Maestro API

All API responses include a `X-Maestro-Version` header with the current Maestro module version.

4.1 Healthcheck

4.1.1 Healthcheck

`GET /healthcheck`

Validates that the app is still up, including the database connection.

- Success Response
  - Code: `200`
  - Content:

```
{
  "healthy": true
}
```

- Error Response

  It will return an error if it failed to connect to the database.

  - Code: `500`
  - Content:

```
{
  "healthy": false
}
```

4.2 Room Management

4.2.1 Ping

`PUT /scheduler/:schedulerName/rooms/:roomName/ping`

This route should be called every 10 seconds and serves as a keep alive sent by the GRU to Maestro.

- Request

```
{
  timestamp: [int]<seconds since epoch>,
  status:    [string]<room-status>
}
```

- Success Response
  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.

  - Code: `422|400`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

  It will return an error if some other error occurred.

  - Code: `500`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.2.2 Address Polling

`GET /scheduler/:schedulerName/rooms/:roomName/address`

This route should be polled by the GRU in order to obtain the room address (host ip and port).

- Success Response

  - Code: `200`
  - Content:

```
{
  "host":  [string]<host ip>,
  "ports": [
    {
      "port": [int]<room port>,
      "name": [string]<port name>
    },...
  ]
}
```

- Error Response

  It will return an error if some error occurred.

  - Code: `500`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.2.3 Room ready

`PUT /scheduler/:schedulerName/rooms/:roomName/status`

This route should be called every time a room is ready to receive a match. You'll need to make sure it is only called after the room has its address.

- Request

```
{
  timestamp: [int]<seconds since epoch>,
  status:    [string]"ready"
}
```

- Success Response
  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.

  - Code: `422|400`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

It will return an error if some other error occurred.

- Code: `500`
- Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.2.4 Match started

`PUT /scheduler/:schedulerName/rooms/:roomName/status`

This route should be called every time a match is started. It'll indicate that this GRU is occupied and is not available for new matches.

- Request

```
{
  timestamp: [int]<seconds since epoch>,
  status:    [string]"occupied"
}
```

- Success Response
  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.
  - Code: `422|400`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

It will return an error if some other error occurred.

- Code: `500`
- Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.2.5 Match ended

`PUT /scheduler/:schedulerName/rooms/:roomName/status`

This route should be called every time a match is ended. It'll indicate that this GRU is no longer occupied and is available for new matches.

- Request

```
{
  timestamp: [int]<seconds since epoch>,
  status:    [string]"ready"
}
```

- Success Response
  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.
  - Code: `422|400`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

  It will return an error if some other error occurred.
  - Code: `500`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.2.6 List Rooms Ordered By Metric

`GET /scheduler/:schedulerName/rooms`

This route returns an array of room ids ordered by metric in ascending order. If metric is "legacy" or "room" it will return available rooms (status ready).

- Optional query parameters
  - *metric*: valid values are "cpu", "mem", "room" and "legacy". Default: "room".
  - *limit*: number of rooms to be returned, must be an int greater than 0. Default: 5.
- Success Response
  - Code: `200`
  - Content:

```
{
  "rooms":  [roomID1, roomID2, roomID3]
}
```

- Error Response

  It will return an error if some error occurred.
  - Code: `500`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

  It will return an error if invalid metric or limit is sent.
  - Code: `400`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.3 Scheduler Management:

4.3.1 Create

`POST /scheduler`

This route creates a scheduler in Maestro using a provided YAML config.

- Request

```
{
  "name": "room-name",
  "game": "game-name",
  "image": "somens/someimage:v123",
  "affinity": "node-affinity",
  "ports": [
    {
      "containerPort": 5050,
      "protocol": "UDP",
      "name": "port1"
    },
    {
      "containerPort": 8888,
      "protocol": "TCP",
      "name": "port2"
    }
  ],
  "limits": {
    "memory": "128Mi",
    "cpu": "1"
  },
  "shutdownTimeout": 180,
  "autoscaling": {
    "min": 100,
    "up": {
      "delta": 10,
      "trigger": {
        "usage": 70,
        "time": 600,
        "threshold": 80
      },
      "cooldown": 300
    },
    "down": {
      "delta": 2,
      "trigger": {
        "usage": 50,
        "time": 900,
        "threshold": 80
      },
      "cooldown": 300
    }
  },
  "env": [
    {
      "name": "EXAMPLE_ENV_VAR",
      "value": "examplevalue"
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "name": "ANOTHER_ENV_VAR",
      "value": "anothervalue"
    }
  ],
  "cmd": [
    "./room-binary",
    "-serverType",
    "6a8e136b-2dc1-417e-bbe8-0f0a2d2df431"
  ]
}
```

- Success Response
  - Code: `201`
  - Content:

```
  {
    "success": true
  }
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.

  - Code: `422|400`
  - Content:

```
  {
    "code":         [string]<error-code>,
    "error":        [string]<error-message>,
    "description": [string]<error-description>,
    "success":      [bool]false
  }
```

  It will return an error if some other error occurred.

  - Code: `500`
  - Content:

```
  {
    "code":         [string]<error-code>,
    "error":        [string]<error-message>,
    "description": [string]<error-description>,
    "success":      [bool]false
  }
```

4.3.2 Delete

`DELETE /scheduler/:schedulerName`

This route deletes a scheduler in Maestro using the scheduler name.

- Success Response

  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if some other error occurred.

  - Code: `500`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.3.3 Update

`PUT /scheduler/:schedulerName`

This route updates a scheduler in Maestro using a provided YAML config.

- Request

```
{
  "name": "room-name",
  "game": "game-name",
  "image": "somens/someimage:v123",
  "ports": [
    {
      "containerPort": 5050,
      "protocol": "UDP",
      "name": "port1"
    },
    {
      "containerPort": 8888,
      "protocol": "TCP",
      "name": "port2"
    }
  ],
  "limits": {
    "memory": "128Mi",
    "cpu": "1"
  },
  "shutdownTimeout": 180,
  "autoscaling": {
    "min": 100,
    "up": {
      "delta": 10,
      "trigger": {
        "usage": 70,
```

(continues on next page)

(continued from previous page)

```
        "time": 600,
        "threshold": 80
      },
      "cooldown": 300
    },
    "down": {
      "delta": 2,
      "trigger": {
        "usage": 50,
        "time": 900,
        "threshold": 80
      },
      "cooldown": 300
    }
  },
  "env": [
    {
      "name": "EXAMPLE_ENV_VAR",
      "value": "examplevalue"
    },
    {
      "name": "ANOTHER_ENV_VAR",
      "value": "anothervalue"
    }
  ],
  "cmd": [
    "./room-binary",
    "-serverType",
    "6a8e136b-2dc1-417e-bbe8-0f0a2d2df431"
  ]
}
```

- Success Response
  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if scheduler to update is not found on DB

  - Code: `404`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

  It will return an error if `:schedulerName` doesn't match name on config

- Code: `400`
- Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

It will return an error if some other error occurred.

- Code: `500`
- Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.3.4 Update Image

`PUT /scheduler/:schedulerName/image`

This route updates a scheduler's image in Maestro using a provided image name. If scheduler already has the image, nothing is done.

- Request

```
{
  "image": "new-image",
}
```

- Success Response
  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if scheduler to update is not found on DB

  - Code: `404`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
```

(continues on next page)

(continued from previous page)

```
  "success":      [bool]false
}
```

It will return an error if image was not sent on body

- Code: `422`
- Content:

```
{
  "code":         [string]<error-code>,
  "error":        [string]<error-message>,
  "description": [string]<error-description>,
  "success":      [bool]false
}
```

It will return an error if some other error occurred.

- Code: `500`
- Content:

```
{
  "code":         [string]<error-code>,
  "error":        [string]<error-message>,
  "description": [string]<error-description>,
  "success":      [bool]false
}
```

4.3.5 Update Min

`PUT /scheduler/:schedulerName/min`

This route updates a scheduler's minimum number of GRUs. If scheduler already has this min, nothing is done.

- Request

```
{
  "min": [int],
}
```

- Success Response
  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if scheduler to update is not found on DB

  - Code: `404`
  - Content:

```
{
  "code":         [string]<error-code>,
  "error":        [string]<error-message>,
  "description": [string]<error-description>,
  "success":      [bool]false
}
```

It will return an error if min was not sent on body

- Code: `422`
- Content:

```
{
  "code":         [string]<error-code>,
  "error":        [string]<error-message>,
  "description": [string]<error-description>,
  "success":      [bool]false
}
```

It will return an error if some other error occurred.

- Code: `500`
- Content:

```
{
  "code":         [string]<error-code>,
  "error":        [string]<error-message>,
  "description": [string]<error-description>,
  "success":      [bool]false
}
```

4.3.6 Status

`GET /scheduler/:schedulerName`

Returns scheduler status and the room count for each status.

- Success Response
  - Code: `200`
  - Content:

```
{
  "game":                [string]<game-name>,
  "state":               [string]<scheduler-state>,
  "stateLastChangedAt": [int]<timestamp when last change happened>,
  "lastScaleOpAt":       [int]<timestamp when last scale happened>,
  "roomsAtCreating":     [int]<number of rooms with creating status>,
  "roomsAtOccupied":     [int]<number of rooms with occupied status>,
  "roomsAtReady":        [int]<number of rooms with ready status>,
  "roomsAtTerminating": [int]<number of rooms with terminating status>
}
```

- Error Response

  It will return an error if scheduler is not found on DB

- Code: `404`
- Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

It will return an error if `:schedulerName` doesn't match name on config

- Code: `400`
- Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

It will return an error if some other error occurred.

- Code: `500`
- Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.3.7 Scheduler Config

`GET /scheduler/:schedulerName?config` `GET /scheduler/:schedulerName/config?version=<version>`

Returns scheduler config. On the new route, when specified version of a scheduler, returns that one.

- Success Response
  - Code: `200`
  - Content:

```
{
  "yaml": [string]<yaml-config>,
}
```

- Error Response

  It will return an error if scheduler is not found on DB

  - Code: `404`
  - Content:

```
{
  "code":         [string]<error-code>,
  "error":        [string]<error-message>,
  "description": [string]<error-description>,
  "success":      [bool]false
}
```

It will return an error if some other error occurred.

- Code: `500`
- Content:

```
{
  "code":         [string]<error-code>,
  "error":        [string]<error-message>,
  "description": [string]<error-description>,
  "success":      [bool]false
}
```

4.3.8 Scale Up

`POST /scheduler/:schedulerName`

Manually scales up the number of rooms of schedulerName.

- Request

```
{
  "scaleup": [int]amount
}
```

- Success Response
  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.

  - Code: `422|400`
  - Content:

```
{
  "code":         [string]<error-code>,
  "error":        [string]<error-message>,
  "description": [string]<error-description>,
  "success":      [bool]false
}
```

  It will return an error if some other error occurred.

  - Code: `500`

- Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.3.9 Scale Down

`POST /scheduler/:schedulerName`

Manually scales down the number of rooms of schedulerName.

- Request

```
{
  "scaledown": [int]amount
}
```

- Success Response
  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.
  - Code: `422|400`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

  It will return an error if some other error occurred.
  - Code: `500`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.3.10 Scale

`POST /scheduler/:schedulerName`

Manually scales the number of rooms to match the number of replicas specified. If there are more rooms than the 'replicas' parameter, there will be a scale down and the number of rooms will be equal 'replicas'. If there are less rooms than the 'replicas' parameter, there will be a scale up and the number of rooms will be equal 'replicas'. Otherwise, nothing is done.

- Request

```
{
  "replicas": [int]replicas
}
```

- Success Response
  - Code: `200`
  - Content:

```
{
  "success": true
}
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.

  - Code: `422|400`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

  It will return an error if some other error occurred.

  - Code: `500`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.3.11 Releases

`GET /scheduler/:schedulerName/releases`

Returns the releases (versions) of the scheduler. A minor release means that the scheduler changed but the pods didn't need to be recreated. Every attribute related only to Maestro and not to the pods does that, e.g.: autoscaling values, forwarder configurations, etc. This means that the pod can have as label version=v1.0 and the scheduler is in version

v1.1. A major relese means that the pods needed to be recreated in order to respect the new scheduler configuration, e.g.: new image, new ports, new env var, new command. In this case, the scheduler will go, for example, from v1.0 to v2.0 and all new pods must have label verion=v2.0.

- Success Response
  - Code: `200`
  - Content:

```
{
  "releases": [{
    "version": <vesion>,
    "createdAt": <created_at>
  }, ...]
}
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.

  - Code: `422|400`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

  It will return an error if some other error occurred.

  - Code: `500`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.3.12 Rollback

`PUT /scheduler/:schedulerName/rollback`

Rollback to a previous version. It gets the config specified or the versionn and executes an update.

- Success Response
  - Code: `200`
  - Content:

```
{
  "version": <version>
}
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.

  - Code: `422|400`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

  It will return an error if some other error occurred.

  - Code: `500`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

4.3.13 Diff

`PUT /scheduler/:schedulerName/diff`

Returns the diff between the configs of two versions of a scheduler. If no version is specified, diff compares the current version with the previous one. If only version v1 is specified, it gets the diff between v1 and the previous one. If v1 and v2 are specified, it gets the diff between this two versions.

- Success Response
  - Code: `200`
  - Content:

```
{
  "version1": <version1>,
  "version2": <version2>,
  "diff": <diff>,
}
```

- Error Response

  It will return an error if the request is invalid or the sent parameters are incorrect.

  - Code: `422|400`
  - Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
```

(continues on next page)

(continued from previous page)

```
  "success":     [bool]false
}
```

It will return an error if some other error occurred.

- Code: `500`
- Content:

```
{
  "code":        [string]<error-code>,
  "error":       [string]<error-message>,
  "description": [string]<error-description>,
  "success":     [bool]false
}
```

CHAPTER 5

CLI

To start the maestro API or Worker you should use one of the commands available in the CLI.

5.1 API

```
Usage:
  maestro start [flags]

Flags:
  -b, --bind string         bind address (default "0.0.0.0")
      --context string      kubeconfig context
      --incluster           incluster mode (for running on kubernetes)
      --kubeconfig string   path to the kubeconfig file (not needed if using --
↪incluster) (default "<homedir>/.kube/config")
  -p, --port int            bind port (default 8080)

Global Flags:
  -c, --config string   config file (default is ./config/local.yaml) (default "./
↪config/local.yaml")
  -j, --json            json output mode
  -v, --verbose int     Verbosity level => v0: Error, v1=Warning, v2=Info, v3=Debug
```

5.2 Worker

```
Usage:
  maestro worker [flags]

Flags:
      --context string      kubeconfig context
      --incluster           incluster mode (for running on kubernetes)
```

(continues on next page)

(continued from previous page)

```
      --kubeconfig string   path to the kubeconfig file (not needed if using --
↪incluster) (default "<homedir>/.kube/config")

Global Flags:
  -c, --config string   config file (default is ./config/local.yaml) (default "./
↪config/local.yaml")
  -j, --json            json output mode
  -v, --verbose int     Verbosity level => v0: Error, v1=Warning, v2=Info, v3=Debug
```

CHAPTER 6

Testing

6.1 Automated tests

We're using Ginkgo and Gomega for testing our code. Since we're making extensive use of interfaces, external dependencies are mocked for all unit tests.

6.1.1 Unit Tests

We'll try to keep testing coverage as high as possible. To run unit tests simply use:

```
make unit
```

To check the test coverage use:

```
make test-coverage-html  # opens a html file
```

or

```
make test-coverage-func  # prints code coverage to the console
```

6.1.2 Integration Tests

TO BE DONE.

We also have a few integration tests that actually connect to maestro external dependencies. For these you'll need to have docker installed.

To run integration tests run:

```
make integration
```

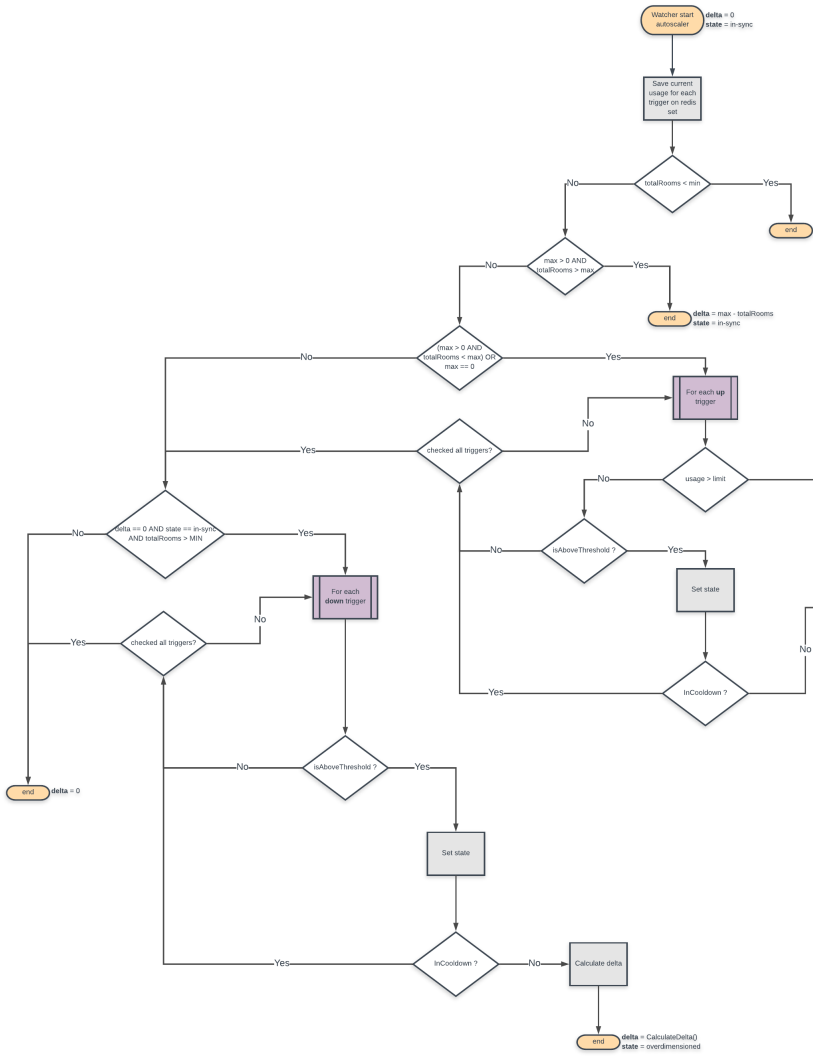
CHAPTER 7

Autoscaling

There is an Autoscaler component in Maestro that starts with each watcher. It follows the scheduler autoscaling configuration present in the configuration YAML of each scheduler.

7.1 Overview

Upscaling and downscaling are handled separately and each accepts a list of **triggers**. These triggers contain all the configuration details that the watchers need to trigger autoscaling when necessary.



The full autoscaling flow is represented below:

Notice that the order of the triggers in the lists is important as autoscaler will look until the first trigger that matches the actual conditions.

7.2 Triggers

Each trigger is composed by the following elements:

1. *type*
2. *usage*
3. *threshold*
4. *time*
5. *limit*

7.2.1 type

Can be **room**, **cpu** or **mem** by default.

- **room**: Scales using the percentage of rooms that are occupied. For upscaling, if more than x% rooms are occupied. For downscaling, if less than x% rooms are occupied. In this case, the number of rooms to scale is given by the following formula:

```
roomsOccupied / ( totalRooms + delta ) = targetUsage / 100
delta = int( Round( (roomsOccupied * 100 - targetUsage * totalRooms) / targetUsage ) )
```

- **cpu** and **mem(memory)**: Scales with the percentage of resources usage per pods. Resource usage is calculated using kubernetes metrics api to get raw usage values and kubernetes client-go to get containers requests values. In this case, the number of rooms to scale is given by the following formula:

```
currentUsage = sum( PodsContainersResourceRawUsage ) / sum(␣
↪PodsContainersResourceRequests )
usageRatio = currentUtilization / targetUsage
targetNumberOfRooms = int( ceil( usageRatio * totalRooms ) )
delta = targetNumberOfRooms - totalRooms
```

7.2.2 usage

It is the target usage in percentage. It is different for up and downscaling and configured via scheduler YAML.

7.2.3 threshold

When the watcher checks state, the autoscaler saves the current usage registered in a redis set. It is called point. For a scale to take place, a percentage of the total amount of points considered must be greater(up trigger) or less(down trigger) than the target usage specified. This percentage is defined by **threshold**.

7.2.4 time

This is the duration of seconds to wait before triggering a scaling. `Time / ( watcher autoScalingPeriod )` gives us the total amount of points that will be considered to define if a scaling need to take place.

7.2.5 limit

It defines a percentage of usage that triggers scaling even if in cooldown period.

7.3 Cooldown

It is the period to wait before running another scaling if it is needed. Cooldown is defined for both up and downscaling.

7.4 Min and Max

These are hard caps for the total number of rooms (ready, occupied and creating) that can exist simultaneously. If Max is set to 0, it means that the scheduler can scale up indefinitely.

7.5 Panic Scale

A **panic scale** happens when a new room is set to occupied and the percentage of occupied rooms is above the limit so it triggers a scale up.

This behaviour is deprecated and disabled by default but can be enabled with `enablePanicScale` flag.

7.6 Requests

Containers resources requests. It is required to define these values in order to use resource triggers(cpu and memory)

7.7 Example autoscaling yaml config:

```
...
requests:
  memory: 1Gi
  cpu: 1000m
autoscaling:
  min: 100                  # minimum amount of GRUs
  max: 1000                 # maximum amount of GRUs
  up:
    metricsTrigger:         # Autoscaling respect the order of the triggers. The␣
↪first that matches will autoscale
      - type: room          # can be room, cpu or memory
        threshold: 80       # percentage of the points that are above 'usage' needed␣
↪to trigger scale up
        usage: 70           # minimum usage (percentage) that can trigger the scaling␣
↪policy
        time: 600           # duration in seconds to wait before scaling policy takes␣
↪place
      - type: mem
        threshold: 80
        usage: 70
        time: 600
    cooldown: 300           # duration in seconds to wait before consecutive scaling
  down:
    metricsTrigger:
      - type: cpu
        threshold: 80
        usage: 50
        time: 900
    cooldown: 300
```

7.8 Creating new autoscaler policies(types)

In order to implement a new autoscaler type, it is required to implement the autoscaler interface:

```
type AutoScalingPolicy interface {
        CalculateDelta(trigger *models.ScalingPolicyMetricsTrigger, roomCount *models.
↪RoomsStatusCount) int
```

(continues on next page)

(continued from previous page)

```
        GetCurrentUtilization(roomCount *models.RoomsStatusCount) float32
}
```

You can find the policies on autoscaler package directory. Then add the new policy to the autoscaler map of policies in the autoscaler instantiation function:

```
func NewAutoScaler(schedulerName string, usageDataSource ...interface{}) *AutoScaler {
        return &AutoScaler{
                AutoScalingPoliciesMap: map[models.
↪AutoScalingPolicyType]AutoScalingPolicy{

                        // legacyPolicy
                        models.LegacyAutoScalingPolicyType: newLegacyUsagePolicy(),

                        // roomUsagePolicy
                        models.RoomAutoScalingPolicyType: newRoomUsagePolicy(),

                        // cpuUsagePolicy
                        models.CPUAutoScalingPolicyType: newCPUUsagePolicy(
                                usageDataSource[0].(kubernetes.Interface),
                                usageDataSource[1].(metricsClient.Interface),
                                schedulerName,
                        ),

                        // memUsagePolicy
                        models.MemAutoScalingPolicyType: newMemUsagePolicy(
                                usageDataSource[0].(kubernetes.Interface),
                                usageDataSource[1].(metricsClient.Interface),
                                schedulerName,
      ),

      // newTypeUsagePolicy
                        models.NewTypeAutoScalingPolicyType: newNewTypeUsagePolicy(
        // Optional dataSources
                                usageDataSource[0].(kubernetes.Interface),
                                usageDataSource[1].(metricsClient.Interface),
                        ),
                },
        }
}
```

And create the new type constant on autoscaler model:

```
type AutoScalingPolicyType string

const (
        // LegacyAutoScalingPolicyType defines legacy usage autoscaling policy type
        LegacyAutoScalingPolicyType AutoScalingPolicyType = "legacy"
        // RoomAutoScalingPolicyType defines room usage autoscaling policy type
        RoomAutoScalingPolicyType AutoScalingPolicyType = "room"
        // CPUAutoScalingPolicyType defines CPU usage autoscaling policy type
        CPUAutoScalingPolicyType AutoScalingPolicyType = "cpu"
        // MemAutoScalingPolicyType defines memory usage autoscaling policy type
  MemAutoScalingPolicyType AutoScalingPolicyType = "mem"
  // NewType defines a new usage autoscaling policy type
        NewTypeAutoScalingPolicyType AutoScalingPolicyType = "newType"
)
```

After that, you can start using your new type on scheduler config:

```
metricsTrigger:
  - type: newType
    threshold: 80
    usage: 50
    time: 900
    limit: 90
cooldown: 300
```

7.9 Best Practices

7.9.1 Upscaling

Maestro upscale can take some time. Hence, it's recommended to use a little lower upscale usage configuration (e.g. 60) to have a good number of unoccupied rooms, in case of a huge load of matches.

7.9.2 Downscaling

For downscaling configuration, it's recommended to use a value lower than the upscaling usage (e.g. 50), in order to prevent an upscaling/downscaling cycle that can keep the scheduler locked for updates.

7.9.3 Access Peaks

When you expect a great income of matches, it's recommended to pre-scale the scheduler by increasing the minimum number of rooms in the scheduler and minimize eventual problems due to slow scales.

CHAPTER 8

Event Forwarders

Event forwarders are pluggable components that forwards events like: RoomReady, RoomTerminated, RoomTerminating, etc... to other services.

A event forwarder is a go native plugin that should be compiled and put into bin folder, it should contain a method `func NewForwarder(config *viper.Viper) (eventforwarder.EventForwarder)` that returns a configured instance of a struct that implements "eventforwarder.EventForwarder".

An example is provided in the plugins/grpc folder, compile it with:

```
go build -o bin/grpc.so -buildmode=plugin plugins/grpc/forwarder.go
```

8.1 Configuration

Then to turn it on, include a config like that in the active config file:

```
forwarders:
  grpc:
    matchmaking:
      address: "10.0.23.57:10000"
    local:
      address: "localhost:10000"
```

In this example, maestro will look for a plugin "grpc.so" in the bin folder and create 2 forwarders from it, matchmaking and local one, each using a different address. Then, every time a room is changing states, all forwarders will be called with infos about the change.

8.1.1 Testing

There's also a route: `/scheduler/{schedulerName}/rooms/{roomName}/playerevent` that can be called like that, for example:

```
curl -X POST -d '{"timestamp":12424124234, "event":"playerJoin", "metadata":{"playerId
↪":"sime"}}' localhost:8080/scheduler/some/rooms/r1/playerevent
```

It will forward the playerEvent "playerJoin" with the provided metadata and roomId to all the configured forwarders.

For the provided plugin, the valid values for event field are: ['playerJoin','playerLeft']

8.2 Responses

All forwarders responses are put in consideration even if they're not returned to the clients. This means that if a forwarder fails the API notifying the event also fails. This behaviour is deprecated and in future versions the forwarder failure/success won't impact any of the notifiers. To achieve this in the current version you can add the `forwardMessage (default: true)` configuration to your scheduler like in this example:

```
forwarders:
  grpc:
    matchamaking:
      enabled: true
      forwardResponse: false
```