# Radar Documentation

***Release 0.0.1b***

**Lucas Liendo**

October 14, 2015

Contents

Radar is a simple and easy to use monitoring tool designed with flexibility and extensibility in mind.

It mainly aims for ease of use, all configuration is handled through YAML files and nearly all options are set by default. A typical installation runs out of the box with minimum intervention.

What Radar is capable to monitor is really up to you. You can write checks in your favourite programming language to monitor anything that you're interested in.

CHAPTER 1

# Contents

## 1.1 Overview

Radar is a monitoring tool that tries to make things simple and easy. Radar is not an advanced monitoring tool and you may not find all features that you may find in other monitoring solutions because Radar's philosophy is based heavily on two ideas :

- Simplicity : Both user-interface and source code are intended to be as simple and clean as possible. The code isn't designed to grow indefinitely mainly because by limiting its scope, its behaviour can be better controlled, tested and understood. The code is designed to be understood by anyone who has some Python and OOP knowledge. Its main ideas are described later in the internals section of this document.
- Extensibility : As you will see later Radar is not focused on any particular resource monitoring. Instead it allows you to integrate any custom designed checks to verify any resource that you want. This does not only applies to IT infrastructure, you could for example read data from a sensor that is attached to a PC or device that is capable of running the Python interpreter.

Once you get the Radar server up and running any number of clients may connect and after a user-defined interval they will be constantly polled. The following diagram shows the big picture how Radar operates : So what can you really do with Radar ? Here are some ideas :

- Notifications : This is a typical use. Notify any of your defined contacts if something is not performing as expected. You can notify by email, sms or maybe put data in a queue and have another process take that responsibility. You could also notify through a real time messaging system like XMPP.
- Graphing tool : Each time you receive data from a check you can add that information to a database (like RRDtool) and then generate graphs. You can get trends of your resources this way.
- Take an advanced action : Suppose that you run a cluster of servers on a cloud-based provider. Then by inspecting the replies of your checks, you can write a plugin to add/remove servers on demand, if you get for example that the amount of requests/sec. that a server is processing are exceeded.

Any of these ideas requires that you write a plugin for it, Radar does not include any built-in functionality and these tasks are delegated to plugins. You can take a look how to develop checks and plugins on the following sections of this document. Don't panic ! A lot of effort has been put to make these tasks as simple as possible, so keep reading the docs.

## 1.2 Minimum requiremients

In order to run, Radar needs the Python interpreter (at least version 2.7) and a few dependencies. These dependencies are already defined in the project, so you don't need to install them manually.

If you have successfully installed the Python interpreter then is highly probable that Radar will run without any platform specific issues. Radar does not depend on any external framework.

Radar (both client and server) should run at least on the following platforms without problems :

- GNU/Linux.
- BSD's.
- Microsoft Windows.
- Darwin / OSX.

## 1.3 Installation

At the moment Radar is only available from its source code from Github. As the project evolves it will be eventually added to PyPI.

PyPI packages will be delivered when the project is assured to be stable. Current Radar's status is ALPHA.

To install Radar from Github (you will need GIT installed on your system) open a terminal or command prompt and run :

```
git clone https://github.com/lliendo/Radar.git
cd Radar
python setup.py install
```

Alternatively you can download Radar as a .zip file, extract it and then run :

```
python setup.py install
```

from the project's root directory.

### 1.3.1 Checking the installation

You can check that Radar was successfully installed by running from the command line :

```
radar-client.py -v
radar-server.py -v
```

This will display the current version of Radar (client and server) installed on your system. If you see these versions then you got both client and server successfully installed, otherwise something went wrong.

By default Radar expects to find configuration files in well-defined places. These directories are platform dependent (you can check these defaults from the platform defaults section of this document).

It is a good practice to use these default directories. Of course if you don't want to use those locations you can change them from the main configuration file.

Radar only takes its main configuration file to be able to run. If you want to override the default main configuration file path you can invoke Radar this way :

```
radar-client.py -c PATH_TO_MAIN_CONFIGURATION_FILE
radar-server.py -c PATH_TO_MAIN_CONFIGURATION_FILE
```

The -c option specifies an alternate main configuration file.

### 1.3.2 Radar setup

Before you start configuring Radar I recommend you to read the documentation as some options may not make full sense. If you've already read the docs then go ahead and start configuring Radar.

Radar comes with two useful scripts to help you configure it the first time.

To configure the server just run :

```
radar-server-config.py
```

This script will ask you for some initial values. For every option you can leave its default (these values are shown in squared brackets) value by pressing Enter.

To configure the client run :

```
radar-client-config.py
```

After you run those scripts the main configuration file gets generated in the path that you chose. Note that the resulting YAML file may not look as tidy as the ones presented in the rest of documentation. This is because the PyYAML library does not care about new lines and does not handle element ordering. Something similar happens on the order in which the options are scanned from the console. You can run these scripts as many times as you want but be aware that if you point to the same output files they'll be completely overwritten.

## 1.4 Platform defaults

Radar supports all platforms where the Python interpreter is able to run. However, Radar internally makes a distinction between Unix like OS's and Microsoft Windows platforms.

### 1.4.1 Unix OS's

Unix OS's have these default values :

Radar server (/etc/radar/server/main.yml) :

```
listen:
    address: localhost
    port: 3333

run as:
    user: radar
    group: radar

log:
    to: /var/log/radar-server.log
    size: 100
    rotations: 5

pid file: /var/run/radar-server.pid
polling time: 300
checks: /etc/radar/server/config/checks
contacts: /etc/radar/server/config/contacts
monitors: /etc/radar/server/config/monitors
plugins: /usr/local/radar/server/plugins
```

Radar client (/etc/radar/client/main.yml) :

```
connect:
    to: localhost
    port: 3333

run as:
    user: radar
    group: radar

log:
    to: /var/log/radar-client.log
    size: 100
    rotations: 5

pid file: /var/run/radar-client.pid
checks: /usr/local/radar/client/checks
enforce ownership: True
reconnect: True
```

### 1.4.2 Windows platforms

Windows platforms have the following default values :

Radar server (C:\Program Files\Radar\Server\Config\main.yml) :

```
listen:
    address: localhost
    port: 3333

log:
    to: C:\Program Files\Radar\Log\radar-server.log
    size: 100
    rotations: 5

polling time: 300
checks: C:\Program Files\Radar\Server\Config\Checks
contacts: C:\Program Files\Radar\Server\Config\Contacts
monitors: C:\Program Files\Radar\Server\Config\Monitors
plugins: C:\Program Files\Radar\Server\Config\Plugins
```

Radar client (C:\Program Files\Radar\Client\Config\main.yml) :

```
connect:
    to: localhost
    port: 3333

log:
    to: C:\Program Files\Radar\Log\radar-client.log
    size: 100
    rotations: 5

checks: C:\Program Files\Radar\Client\Config\Checks
reconnect: True
```

## 1.5 Server configuration

Radar is simple and flexible in the way it defines different kind of components (checks, contacts and monitors). It uses YAML for all of its configuration.

When Radar starts it expects a main configuration file (this applies to both client and server) as input. From that configuration file, further locations are read to find checks, contacts, monitors definitions and plugins.

You're going to read a long document where the most important concepts of Radar are described. Take a cup of your favourite beverage and read it carefully. Once you read it you will find that configuring Radar is easy. Try not to skip any sections because every part of it has something to say. By the end of this document you'll have solid knowledge on how to configure Radar and not going into trouble on the way.

### 1.5.1 Main configuration

The main configuration governs general aspects of the Radar server. We'll take a look at a full configuration file and describe every available option :

```
listen:
    address: 192.168.0.100
    port: 3333

run as:
    user: radar
    group: radar

log:
    to: /tmp/radar/logs/radar-server.log
    size: 10
    rotations: 3

polling time: 300
pidfile: /tmp/radar-server.pid
checks: /tmp/radar/server/checks
contacts: /tmp/radar/server/contacts
monitors: /tmp/radar/server/monitors
plugins: /tmp/radar/server/plugins
```

- listen : The listen options specifies the address and port number where Radar server is going to listen for new clients. At the moment only IPv4 addresses are supported. The default values are to listen on localhost and port 3333.
- run as : On Unix platforms this option tells Radar the effective user and group that the process should run as. This is a basic security consideration to allow Radar to run with an unprivileged user. The specified user should not be a able to login to the system. The default user and group is radar. This option does not apply to Windows platforms.
- polling time : This tells Radar the frequency of the execution of checks. It is expressed in seconds. By default Radar will poll all its clients every 300 seconds (that is five minutes). You're not allowed to specify values under one second. Fractions of a second are allowed so you can poll your clients let's say every 10.5 seconds.
- log : Radar will log all of its activity in this file. So if you feel that something is not working properly this is the place to look for any errors. Note that in the example there are two additional options : size and rotations. They indicate the maximum size (in MiB) that a log should grow, when its size goes beyond that amount then is rotated (backed up) and new logs are written to a new file. By default Radar sets a maximum of 100 MiB for the log file and rotates it at most 5 times.

- pid file : On Unix platforms this file holds the PID of the Radar process. When Radar starts it will record its pidfile here and when it shuts down this file is deleted. Pid files are not recorded on Windows platforms.
- checks, contacts, monitors, plugins : All these set of options tell Radar where to look for checks, contacts, monitors and plugins. They're just directory names. Radar does not impose on you any layout on these directories. This means that you are free to put configurations on any file on any directory below the ones you defined. The only requirement is that you are not allowed to mix for example contacts or contact groups with other components such as checks or check groups. Every directory is designed to contain specific elements. These set of directories are platform dependant.

Let's now take a look at a minimum configuration. Every platform has a default configuration and the options that are different from one another are the ones that are platform dependant.

```
listen:
    address: 192.168.0.100

polling time: 60
```

As you can see from the example most of the options are not defined. They are not really missing, if you don't specify an option it will take its default value. This means that you only need to worry about the options you want to modify. The above example will poll clients every minute and listen on 192.168.0.100:3333.

### 1.5.2 Checks configuration

As commented above all checks and check groups definitions are expected to be found in the checks defined directory.

How checks are defined ? Each check is defined as follows :

```
- check:
    name: CHECK NAME
    path: PATH TO CHECK
    args: CHECK ARGUMENTS
```

Let's review each parameter of a check definition :

- name : Each check must be uniquely identified. This is the purpose of the name parameter, it acts as a unique identifier. You can use whatever name you like. This parameter is mandatory.
- path : The full filesystem path to the check. If this path is not absolute then the check is looked up in the client's defined check directory. This parameter is mandatory.
- args : This parameter is used to specify any additional arguments that you need to pass to the check. This parameter is optional.

Let's now move on defining check groups. Check groups can be defined in two different ways, let's see the first one :

```
- check group:
    name: CHECK GROUP NAME
    checks:
        - check:
            name: CHECK NAME
            path: PATH TO CHECK
            args: CHECK ARGUMENTS
```

You define a check group by giving that group a name and a set of checks that make up that group. This allows you to reference a check group later on when you define monitors. Check groups are useful because you define only once a group and then use it in any number of monitors.

Let's now take a look at a second way of defining a check group :

```
- check:
    name: CHECK NAME
    path: PATH TO CHECK
    args: CHECK ARGUMENTS

- check group:
    name: CHECK GROUP NAME
    checks:
        - check:
            name: CHECK NAME
```

In this example we've defined a check first and referenced it later from a check group. This is perfectly valid and is actually a very convenient way to define check groups. Why ? Let's suppose that you have two or more check groups that are very similar but some of them perform additionally other checks, then by defining checks individually and referencing them allows you to define checks once and use them in as many groups as you want making the overall configuration shorter and easier to understand. Note that the check definition could also had been defined after the check group because Radar does not care about definition order. Being that said the above configuration is equal to :

```
- check group:
    name: CHECK GROUP NAME
    checks:
        - check:
            name: CHECK NAME

- check:
    name: CHECK NAME
    path: PATH TO CHECK
    args: CHECK ARGUMENTS
```

Here's a fragment of how a real configuration might look like :

```
- check group:
    name: Basic health
    checks:
        - check:
            name: Uptime
            path: uptime.py
            args: -S 300

        - check:
            name: Ram usage
            path: ram-usage.py
            args: -O 0,1000 -W 1000,1900


- check group:
    name: Disk usage
    checks:
        - check:
            name: Disk usage (/)
            path: disk-usage.py
            args: -p / -O 0,8 -W 8,10 -u gib

        - check:
            name: Disk usage (/home)
            path: disk-usage.py
            args: -p /home -O 0,100 -W 100,150 -u gib
```

Some final notes on defining checks (this actually applies to the overall configuration) :

- Radar expects at least one check or check group to exist in the overall configuration. Otherwise, why use Radar if you don't want to check at least one resource ?
- Checks and check groups are allowed to be repeated and Radar won't complain at all. However there are no guarantees at all which of the repeated check or check groups Radar will keep. The rule is that you must not duplicate check or check groups names.
- As stated before the order of definition does not matter because Radar will first build all of its checks and then proceed to build all the check groups. The same applies for contacts and contact groups.
- If you have a relatively big configuration then it might be useful to split it among different files and in some cases among directories. Remember that Radar does not impose you any restrictions on this.

### 1.5.3 Contacts configuration

If you understood how checks and checks groups are defined then defining contacts and contact groups is exactly the same !

Here's an example of a contact definition :

```
- contact:
    name: CONTACT NAME
    email: CONTACT EMAIL
    phone: CONTACT PHONE NUMBER
```

- name : Each contact must be uniquely identified. This is the purpose of the name parameter, it acts as a unique identifier. You can use whatever name you like. This parameter is mandatory.
- email : The email of the contact you're defining. Radar won't check at all if the defined email address is valid, so be careful ! This parameter is mandatory.
- phone : This is the phone number of the contact. Radar won't check if this is a valid phone number. This parameter is optional.

Let's see a contact group definition :

```
- contact group:
    name: CONTACT GROUP NAME
    contacts:
        - contact:
            name: CONTACT NAME
            email: CONTACT EMAIL
            phone: CONTACT PHONE NUMBER
```

Compare the above definitions (against checks and check groups). You'll realize that they are almost identical, of course the identifiers for each component are different but the same idea remains : you can compose contact groups as you like and reference contacts from any contact group.

Here's a fragment of how a real configuration might look like :

```
- contact group:
    name: Sysadmins
    contacts:
        - contact:
            name: Hernan Liendo
            email: hernan@invader
        - contact:
            name: Javier Liendo
            email: javier@invader
```

There is one little difference between checks and contacts definitions. In some scenarios it might not be needed to notify any contact at all, so Radar allows you to leave contacts empty, in other words defining contacts and contact groups is completely optional.

### 1.5.4 Monitors configuration

Once you have defined all your contacts and checks the last step is to define monitors. Monitors are the way to tell Radar which hosts to watch, what to check and who notify.

Let's walk through a real example :

```
- monitor:
    hosts: [localhost, 192.168.0.101 - 192.168.0.200]
    watch: [Basic health, Disk usage]
    notify: [Sysadmins]
```

The above example is telling Radar to monitor localhost and all hosts that are in the 192.168.0.101 - 192.168.0.200 range and to check for Basic health, Disk usage and to notify Sysadmins. So to define monitors you basically have :

```
- monitor:
    hosts: [HOSTNAME | IP | IP RANGE, ...]
    watch: [CHECK | CHECK GROUP, ...]
    notify: [CONTACT | CONTACT GROUP, ...]
```

- hosts : There are three different way to specify hosts. You can specify a single host by its IPv4 (this if the preferred way) or by its hostname. The last way to define hosts is using an IPv4 range. This is useful for example if you want to run the same checks on a set of hosts. Ranges are specified by its start, a hyphen and its end ip. The initial and ending hosts are included in the range.
- watch : This is a list of checks or check groups to be run on the monitored hosts. You only need to reference previously defined checks or check group names.
- notify : Same as above but for contacts. You need to reference a list of previously defined contacts or contact groups.

Note that the hosts, watch and notify parameters are defined within squared brackets. Don't forget this when defining monitors ! This is the only place where we use a list (more precisely a YAML list) of elements.

You can include as many monitors as you want on each file. There are no restrictions. You need to be careful when you reference checks and contacts in the monitors definition because Radar will not validate the referenced checks and contacts. This means that if you reference a contact, contact group, check or check group that does not exist Radar won't complain. All references in monitors are case sensitive so you also need to be aware about this, the best practice to avoid this kind of issue is to stick to a rule (e.g. always lower case references, camel case, etc).

You may be wondering under which conditions Radar knows if it should notify its contacts. The Radar core does not handle (and does not care) this, but plugins might do. Every time a Radar client replies the server this information is passed to all defined server plugins. If you have a notification plugin installed (e.g. an email notification plugin) it will probably inspect the current and previous status of a check to decide if it should notify the affected contacts.

Don't worry if you don't want to write a Radar plugin (you don't have to, although you're encouraged to at least understand how a plugin works and how it should be designed).

### 1.5.5 Plugins configuration

Radar server relies on plugins to perform certain actions. For example assume that you want to notify your contacts by SMS and you also want to be able to store all your checks data to a relational databse. So it might be perfectly reasonable to ask yourself how to do that with Radar.

Radar does not provide any built-in mechanisms to do these kind of things because that responsability is left to plugins. For the moment we're not going to describe how to write a plugin but how to install them.

As described previously there is one plugin directory defined in the main configuration file. This directory holds all the plugins managed by Radar. How is the layout of this directory ? If you've read previous sections you noticed that you have full freedom to layout monitors, checks and contacts directories. This is not the case for the plugins directory.

Let's assume that your plugins directory is : /tmp/Radar/server/plugins. Then you have a bunch of plugins you want install. Simply copy all of them to that directory.

The layout of the plugins directory might look something like this :

```
/tmp/Radar/server/plugins
    /some-plugin
        /__init__.py
    /another-plugin
        /__init__.py
        /another-plugin.yml
    ...
```

Every plugin must be contained within one directory below the defined plugins directory. Some plugins might contain configurations as well (from the above example 'another-plugin' seems to have its own YAML configuration file). Check each plugin's documentation to figure out the scope of a plugin and how can you adjust it to fit your needs.

## 1.6 Client configuration

If you understood how a Radar server is configured then configuring a Radar client is even easier (mainly because you don't define elements such as contacts, checks, monitors and plugins).

### 1.6.1 Main configuration

The main configuration governs general aspects of the Radar client. We'll take a look at a full configuration file and describe every available option (this time we will setup a Windows Radar client) :

```
connect:
    to: 192.168.0.100
    port: 3333

log:
    to: C:\Radar\Client\radar-client.log
    size: 10
    rotations: 3

checks: C:\Radar\Client\checks
enforce ownership: False
reconnect: False
```

- connect : This option tells Radar client where to connect to. At the moment only IPv4 addresses are supported. By default it tries to connect to localhost port 3333.
- run as : On Unix platforms this option tells Radar the effective user and group that the process should run as. This is a basic security consideration to allow Radar to run with an unprivileged user. The specified user should not be a able to login to the system. The default user and group is radar. This option does not apply to Windows platforms.
- log : Radar will log all of its activity in this file. So if you feel that something is not working properly this is the place to look for any errors. Note that in the example there are two additional options : size and rotations.

They indicate the maximum size (in MiB) that a log should grow, when its size goes beyond that amount then is rotated (backed up) and new logs are written to a new file. By default Radar sets a maximum of 100 MiB for the log file and rotates it at most 5 times.

- pid file : On Unix platforms this file holds the PID of the Radar process. When Radar starts it will record its pidfile here and when it shuts down this file is deleted. Pid files are not recorded on Windows platforms.
- checks : This is the location where all your checks are stored. Every time a Radar client receives a CHECK message from the server all checks are first looked up here if a relative path was given, otherwise it will be looked up by its absolute path. You can lay out this directory as you wish.
- enforce ownership : On Unix platforms if this option is True then every time the Radar client has to execute a check it will first verify that the user and group of any check matches the one defined in the run as option. If the user and group does not match and error is generated and the check won't run. On Windows platforms it will only check for the user. By default this option is set to True.
- reconnect : This option specifies the behaviour of the client when a Radar server goes down. If set to True and if the Radar server stops working the client will keep retrying to connect to the server. By default this option is set to True.

As usual you can leave out almost every option to its default value. A minimum Radar client configuration file might look like this :

```
connect:
    to: 192.168.0.100
```

All remaining options take default values. Check these defaults on the platform defaults section.

## 1.7 Checks development

In this section we will explain how checks should be developed and how Radar runs them. As stated before checks can be programmed in your favourite language.

### 1.7.1 Introduction

Checks are the way Radar monitors any given resource. Radar only cares how you return the output of your checks, it will expect a valid JSON containing one or more of the following fields :

- status : This field is mandatory. It must be equal to any of the following string values : OK, WARNING, SEVERE or ERROR.
- details : This is an optional field. You can add details about how your check performed. Radar expects this value to be a string.
- data : This is an optional field. You can put here any data related to your check. Radar does not expect any particular format for this field.

So, let's assume you have an uptime check. So the minimum output for this check would be the following JSON :

```
{"status": "OK"}
```

Radar is case insensitive when it reads a check's output so it does not care how you name the fields (as long as you include them Radar won't complain), so this JSON is also valid :

```
{"Status": "Ok"}
```

Any other combination of upper and lower characters is also valid. Now suppose you want to be more verbose on this check, then you might want to add some details :

```
{
    "status": "OK",
    "details": "0 days 6 hours 58 minutes"
}
```

Details will be stored (along with status and data) in their respective checks on the server side, and all your plugins will have visibility on these fields. Now, let's assume that your check also returns data to be processed by any of the plugins on the server side, so your JSON might look something like this :

```
{
    "status": "OK",
    "data": {
        "uptime": 25092,
        "name": "uptime"
    },
    "details": "0 days 6 hours 58 minutes"
}
```

As mentioned before, there is no specific format that you have to comply on the data field, is completely up to you how to lay it out. You can include as much data as you want and in any desired way.

### 1.7.2 Guidelines

Currently Radar checks are executed sequentially, that is one after the other (this is expected to change on a future release), so some care must be taken when developing them. Here are some tips and advices on how to write good checks :

- The order of the checks execution is irrelevant. That means that check Y must not depend on previous execution of check X.
- Checks should check one and only one resource. If you're checking the load average, then just check that and not other things like free memory or disk usage. For those write 2 other checks.
- Checks should run as fast as possible or fail quickly. As checks run sequentially if one check lasts 10 seconds to complete, then subsecuent checks will begin executing after 10 seconds.

  Always fail fast if the resource you're checking is not available for any reason. For example if you're checking the status of your favourite SQL engine and for whatever reason is not running or does not allow you to get its status, then return an ERROR status explaining the reason and don't keep retrying to get its status over and over again.
- Checks should be platform independent. It is nice to just write a check once and then have it running on as many platforms as possible. This promotes reusability.
- Test your checks ! You're encouraged to write unit tests on all of your checks. This way you make sure that checks behave and fail as expected.

### 1.7.3 Examples

If you're looking for some real examples you can take a look at this repository. In there you'll will find some basic but useful checks (written in Python) that allows you to monitor :

- Disk usage.
- Ram usage.
- Uptime.
- Process status.

They have been designed to run on as many platforms as possible. They rely on the excellent psutil module.

## 1.8 Plugins development

We're now going to describe plugin development. Although you may think that this might be a complex task, a lot of effort has been put in the design of this part of Radar so you can easily write a plugin. You will need to at least understand a little of Python and object oriented programming.

Even if you're not proficient with Python or object oriented programming keep reading and decide by yourself if writing a Radar plugin is a difficult task.

### 1.8.1 Introduction

When you first launch Radar all plugins are instantiated, that is for every plugin that Radar finds in its plugins directory it tries to create an object. This isn't just any object, it has to comply somehow with what Radar expects to be a valid plugin. If Radar instantiates a certain plugin without problems then it proceeds to configure it. After it has been configured it is appended to a set of plugins.

When the server receives a check reply every plugin is sequentially invoked passing it some information. That's all Radar does, from that point (when your plugin receives a check reply) you have partial control on what is done. When all plugins finish processing a certain reply, full control is regained by Radar. This process repeats indefinitely until of course you shut down Radar.

We've just described how Radar processes plugins. We're now going to take a look at how a minimal plugin is written and what considerations should be taken at development time.

Let's take a look at a minimal plugin and describe the key points.

Take a look a this piece of Python code :

```
from radar.plugin import ServerPlugin


class DummyPlugin(ServerPlugin):

    PLUGIN_NAME = 'Dummy plugin'

    def on_start(self):
        self.log('Starting up.')

    def on_check_reply(self, address, port, checks, contacts):
        self.log('Received check reply from {:}:{:}.'.format(address, port))

    def on_shutdown(self):
        self.log('Shutting down.')
```

As explained before, a Radar plugin needs to comply with certain requirements. In first place every plugin must inherit from the ServerPlugin class. You achieve that by importing the ServerPlugin class and creating a new class and inheriting from ServerPlugin. This is achieved in the first two lines of the above example.

Every plugin must have a name. We define this in the PLUGIN_NAME class attribute. Every plugin is uniquely identified by its name and a version. If you don't specify a version then it defaults to 0.0.1. To define a version you only need to overwrite the PLUGIN_VERSION class attribute with the desired value. In this exaple we only defined the plugin name.

The example shows three methods. The on_start() method is invoked by Radar when the plugin is initialized, so if you want to define any instance attributes or acquire resources, this is one place to do that.

In a similar way the on_shutdown() method is called when Radar is shutdown, this method's purpose is to gracefully release any resources that you might have acquired during the life of the plugin.

We have only one remaining method: on_check_reply(). This is where the action takes place and is your entry point to perform any useful work. For every reply that the Radar server receives you'll get :

- address : The IP address of the Radar client that sent the check reply. This is a string value.
- port : The TCP port of the Radar client that sent the check reply. This is an integer value.
- checks : A Python list containing Check objects that were updated by the server. This list will always respond to a given monitor, that means that the list of checks you got belongs to one and only one monitor.
- contacts : A Python list containing Contact objects that were related due to the replied checks. This list will always respond to a given monitor, that means that the list of contacts you got belongs to one and only one monitor.

In this minimal example we're basically doing nothing, just recording a few things to the Radar log file using the log() method. A Radar plugin is just a Python class where you can code anything you want.

If you want to verify this small plugin, then :

1. Create a directory called dummy-plugin (or name it as you want).
2. Create an __init__.py file inside this directory and copy the above code to it.
3. Move the directory you created in step one to your Radar server's plugins directory.
4. If Radar is already running then you'll need to restart it.
5. Make sure you get a new entry in the log every time a check reply arrives.

Is that all you need to know to develop a plugin ? Basically yes, but there is one more feature that can be extremely useful in some cases. Let's say you want to allow your users configure your plugin, that is let your users modify certain parameters of your plugin. If you've just wrote a plugin that connects to a database to insert data then it is not a good practice to modify the database parameters directly from the plugin code. Radar plugins come with a YAML mapper for free.

What does it do ? Very simple : given a YAML filename it will map it to a Python dictionary. This way you only specify the filename of your configuration file, set the values that you need in that file and then retrieve them from a dictionary. The only requirement is that this file must be in your plugin directory !

To use it simply set the PLUGIN_CONFIG_FILE class attribute with the configuration filename and that's it. How do you read those values ? Easy again, just access the config dictionary. Let's see an example. Suppose you want to proxy every reply to another service using a UDP socket.

Given this YAML file (called udp-proxy.yml) :

```
forward:
    to: localhost
    port: 2000
```

Let's adjust our initial example :

```
from socket import socket, AF_INET, SOCK_DGRAM
from json import dumps
from radar.plugin import ServerPlugin


class UDPProxyPlugin(ServerPlugin):

    PLUGIN_NAME = 'UDP-Proxy'
    PLUGIN_CONFIG_FILE = ServerPlugin.get_path(__file__, 'udp-proxy.yml')
    DEFAULT_CONFIG = {
```

```
        'forward': {
            'to': '127.0.0.1',
            'port': 2000,
        }
    }

    def _create_socket(self):
        fd = None

        try:
            fd = socket(AF_INET, SOCK_DGRAM)
        except Exception, e:
            self.log('Error - Couldn\'t create UDP socket. Details : {:}.'.format(e))

        return fd

    def _disconnect(self):
        self._fd.close()

    def on_start(self):
        self._fd = self._create_socket()

    def _forward(self, address, checks, contacts):
        serialized = {
            'address': address,
            'checks': [c.to_dict() for c in checks],
            'contacts': [c.to_dict() for c in contacts],
        }

        payload = dumps(serialized) + '\n'
        self._fd.sendto(payload, (self.config['forward']['to'], self.config['forward']['port']))

        return payload

    def on_check_reply(self, address, port, checks, contacts):
        try:
            self._forward(address, checks, contacts)
        except Exception, e:
            self.log('Error - Couldn\'t forward data. Details : {:}.'.format(e))

    def on_shutdown(self):
        self._disconnect()
```

Ok, now we have a useful plugin. Every time we receive a reply we simply forward it using a UDP socket. Note in this example that I've set the PLUGIN_CONFIG_FILE to hold the filename of the YAML (udp-proxy.yml in this case) and that I use the values that were read from that file in the _forward() method. Also note the use of the get_path() static method to properly reference the YAML file and that I convert every check and contact to a dictionary before serializing and sending the data. The to_dict() method dumps every relevant attribute of each object to a Python dictionary.

Now take a look at the DEFAULT_CONFIG class attribute. This class attribute allows you to set default values for your plugin configuration provided that a user does not set a certain parameter. Radar (internally) will merge the values read from the file and those found in the DEFAULT_CONFIG class attribute. Setting this dictionary is completly optional. This can be very useful for example if a user forgets to create a configuration file for your plugin, by using a default config you make sure that at least your plugin won't fail due to a missing configuration.

To get this example running follow the same steps we described for the DummyPlugin and also create a file named udp-proxy.yml that contains the YAML commented above. Don't forget to put this file inside the same directory where

__init__.py is.

If you want to see these replies you'll probably need a tool like Netcat. If you indeed have Netcat installed on your system then open up a console and run :

```
nc -ul 127.0.0.1 2000
```

The above command will capture and display UDP datagrams destined for localhost port 2000.

Before we end up this section you may be wondering : How should I use the checks and contacts lists in the on_check_reply() method ?

Radar has (internally) among many abstractions two that you will use directly in any plugin : Contact and Check. Whenever you get a reply you get a list that contains contact objects and another list that contains check objects.

Contact and Check objects have some attributes that you can read to perform some work. For example : every contact object contains a name, an email and optionally a phone number. The following piece of code shows how to read any useful value (both from a contact and a check) :

```
from radar.plugin import ServerPlugin


class TestPlugin(ServerPlugin):

    PLUGIN_NAME = 'Test plugin'

    def on_check_reply(self, address, port, checks, contacts):
        """ Accesing properties of a check and contact object """

        """ Contact properties. """
        contact_name = contacts[0].name
        email = contacts[0].email
        phone = contacts[0].phone

        """ Check properties. """
        check_name = check[0].name
        path = check[0].path
        args = check[0].args
        details = check[0].details
        data = check[0].data
        current_status = check[0].current_status
        previous_status = check[0].previous_status
```

Note that in the above example we're only inspecting the first contact and check. Remember that you always receive two lists, so you may need to iterate them in order to achieve your plugin's task.

One last thing. If you inspect the current_status and the previous_status attributes of a check you'll notice that both of them are integers. If you need to convert those values to their respective names, here's how to do that :

```
from radar.plugin import ServerPlugin
from radar.check import Check


class TestPlugin(ServerPlugin):

    PLUGIN_NAME = 'Test plugin'

    def on_check_reply(self, address, port, checks, contacts):
        """ Converting check reply status values to string codes. """
```

```
        current_status = Check.get_status(check[0].current_status)
        previous_status = Check.get_status(check[0].previous_status)
```

The conversion is done using the static Check.get_status() method. Note that I've also imported the Check class in the second line of the example. Now current_status and previous_status hold any of the valid string codes that a check can return (OK, WARINING, SEVERE or ERROR).

### 1.8.2 Guidelines

All of the considerations taken to develop checks also apply to plugins. So if in doubt review those guidelines in the checks development section.

Also note that Radar expects to find a unique plugin class per plugin directory. It is a requirement that this class to be present only in the __init__.py file in that directory. Despite this minor limitation you're allowed to code in as many different directory/files inside the plugin directory as you want.

For example, assuming that you wrote the ProxyPlugin described above then, you could have the following file hierarchy :

```
/ProxyPlugin
    /__init__.py
    /proxy.yml
```

If your ProxyPlugin also depended on more modules then you could had :

```
/ProxyPlugin
    /__init__.py
    /proxy.yml
        /another_module
            /__init__.py
            /another_file.py
```

### 1.8.3 Example

If you still want to see a more elaborate example (actually something useful, right ?) then you can take a look to an email notifier plugin here. This plugin will notify its contacts when a check has any of its status (current or previous) distinct from OK.

## 1.9 Radar internals

Radar has been carefully designed to its keep base code clean and understandable, so everyone can take a look at its internals (and hopefully play with the code).

This section of the documentation tries to expose the main ideas that were implemented to make this project possible. We'll not describe every single detail because that would take a huge amount of time and you'll get bored. Instead I've decided to describe few things as possible and try to reflect why a decision was taken that way. Also consider that like everybody else, I make mistakes and no perfect software design exists and Radar is a long way from achieving that.

### 1.9.1 Overview

Radar is designed to be a small tool, its core isn't intended to grow indefinitely besides some currently lacking features. The reason behind this is that a tool that is small and controlled in its size and its objectives is easier to understand

and does its work better than an all-problem-solving solution. This also has a downside : a small tool may not offer advanced or complex features. Radar's main goal is to be a simple and easy to use tool hence the reason why you might not find as many features as other solutions may offer.

Radar makes use of object oriented programming, every component is modeled using a class. Some few classes make use of mixins and all errors are handled through exceptions. Radar also makes heavy use of list comprehensions across the project.

If you take a fast look to the code you'll realize that almost every method is only a few lines long. Every class is intended to perform a specific task and each method solves a concrete piece of that task. The result is that you won't find complex or twisted code and reading any piece of code and get the idea of what is doing should take little time. The code mostly lacks comments, the reason for this is that the code intends to be self-describing (care has been taken to make classes and methods describe and reflect their intentions). Radar tries to stick to this rule.

### 1.9.2 Project layout

Radar has the following project structure :

```
/Radar
    /docs                # Includes project's documentation in reStructuredText.
                         # Sphinx is used for documentation generation.

    /scripts             # Launch scripts of Radar server and client.
                         # Configuration scripts for Radar server and client.

    /tests               # Project's tests.

    /radar
        /check           # Check and CheckGroup abstractions.
        /check_manager   # CheckManager governs check execution.
        /class_loader    # Loading mechanism for plugins.
        /client          # Main RadarClient abstraction.

        /client_manager  # ClientManager handles Radar clients when connect,
                         # disconnect and send replies.

        /config          # Includes builders that handle initializations of
                         # both Radar client and server.

        /contact         # Contact and ContactGroup abstractions.

        /initial_setup   # Includes facilities to configure Radar after
                         # it has been installed.

        /launcher        # Includes classes that fire up both Radar server
                         # and client.

        /logger          # Logging services.
        /misc            # A few helper classes mainly used by Radar server.
        /monitor         # Monitor abstraction.

        /network         # Low level network facilities to handle both
                         # server and clients. Different platform specific
                         # network monitors are found here.

        /platform_setup  # Platform specific configuration and setup.

        /plugin          # Plugin and PluginManager classes work each other
```

```
                    # closely to allow plugin functionality.

    /protocol       # Low level network protocol that Radar uses for
                    # communicating between server and clients.

    /server         # Main RadarServer abstraction.
```

### 1.9.3 Initialization

Both Radar client and server go through almost the same steps before going into operational mode. When Radar (client or server) is fired up it instantiates a launcher (RadarClientLauncher for the client and RadarServerLauncher for the server) and immediately calls its run() method.

From that point a three phase initialization takes place :

1. First the command line is processed. This is done in the RadarLauncher class. After this, objects and configurations are read from the main configuration file and alternate files in the case of the server are parsed and processed.
2. Client and server proceed to define, create and configure threads.
3. Finally threads are launched.

After all threads are successfully launched client and server break away and start performing completely different tasks.

### 1.9.4 Operational overview

Both Radar client and server operate in an event triggered fashion and make use of threads to distribute the workload. If you look at the code of the RadarServer and RadarClient classes you'll find methods called 'on_something'. Every time a network event occurs it is reflected in any of those methods. The heart of Radar is two abstract classes : Client and Server which can be found under the network module. The Client and Server classes operate in a very similar way despite being different from the way they handle network sockets.

The network module also provides some network monitors that are platform dependent. Before Radar server goes into operational mode it tries to select the best multiplex i/o method available. In any case if the platform can't be detected or an efficient multiplexing method cannot be found Radar will fall back to the SelectMonitor (which relies on the select system call). The currently supported multiplexing strategies are : select, poll, epoll and kqueue.

Radar's client and server also operate in a non-blocking way. Its main threads loops are iterated constantly every 200 milliseconds. This prevents any single client from blocking the server indefinitely due to a malformed or incomplete network message. Also this mechanism is used as an easy workaround to gracefully terminate threads : one thread Event is shared among all defined threads, when this thread event is stopped the condition of the loop does not hold and the threads successfully end.

### 1.9.5 Server operation

The main work of the server is split across three main threads :

- RadarServer.
- RadarServerPoller.
- PluginManager.

RadarServer :

This thread is responsible for accepting clients and receiving replies from them. A client is only accepted if it is defined in at least one monitor and is not duplicated (that is, if the same client isn't already connected).

Once a client is accepted it is registered within the ClientManager. The ClientManager acts as proxy that talks directly to all defined monitors. Every monitor internally knows if it has to accept a client when it connects, if it is indeed accepted then a copy of the checks and contacts is stored along with the instance of that client. This copy is needed because more than one client may match against the same monitor.

The reverse process applies when a client disconnects, the RadarServer unregisters that client and the connection is closed.

When a client sends a reply is it also initially processed by the ClientManager. The reason for this is that we need to get a list of checks and contacts that are affected by such reply. These two lists of objects are later on transferred to the PluginManager to be processed by any defined plugins.

RadarServerPoller :

This is the simplest thread. Every N seconds it simply asks the ClientManager to poll all of its monitors. The existence of this thread is that it makes sense to have a different abstraction that decides when its time to poll the clients. If this work would have been done in the RadarServer we would be mixing asynchronous (network activity) and synchronous (wait a certain amount of time) events making the overall design more complex to both understand and work with.

PluginManager :

As its name indicates, this is the place where all plugins are executed and controlled. Whenever the RadarServer receives a reply from a client and after little processing a dictionary containing all relevant plugin data is written by the RadarServer to a queue that both RadarServer and PluginManager share, this is the mechanism of communication between those objects. The PluginManager quietly waits for a new dictionary to arrive from this queue, when it does it disassembles all parameters and performs object id dereferencing of two lists that contain the affected checks and the related contacts. This dereferencing is possible because threads share the same address space. This solution seems more elegant and effective than re-instantiating those objects from their states. After this pre-processing every plugin's run method is called with appropiate arguments. If a plugin does not work properly all exceptions are caught and registered in the Radar's log file.

### 1.9.6 Client operation

The client relies on two threads :

- RadarClient.
- CheckManager.

RadarClient :

This thread is responsible for receiving and replying messages from the Radar server. For every message received the message is desearialized and written to a queue (that is shared with the CheckManager). Both RadarClient and CheckManager actually share two queues to support bidirectional communication between threads. One queue is used to write checks that need to be executed, the other is used to read the results of those executions.

In case the Radar client is unable to connect to the Radar server it will wait a certain amount of time and try to reconnect again. This is repeated indefinitely if the reconnect option is set to True. It will try to connect after 5, 15 and 60 seconds (cyclically). This option is useful because after updating the Radar's server configuration you need to restart it and all connections are lost. Radar currently does not provide a reload mechanism.

CheckManager :

Whenever a CHECK message is received by the RadarClient thread and after little processing is immediately sent to the CheckManager. When the check information is received the CheckManager proceeds to instantiate a bunch of Checks (depending on the platform running it may instantiate a UnixCheck or a WindowsCheck) and finally executes

them sequentially. Every check's output is collected and verified (the CheckManager makes sure that the Check didn't blow up and that a valid status was returned). It also discards all fields that are not relevant (it will only keep the status, details and data fields of the returned JSON).

Once the outputs have been collected they're sent back to the RadarClient through the other queue and RadarClient sends those results back to the RadarServer.

### 1.9.7 Network protocol

Radar client and server use TCP for all of its communications. Here is the network protocol that is used by Radar :

| TYPE | OPTIONS | PAYLOAD SIZE | PAYLOAD |
|---|---|---|---|

- TYPE (1 byte) : Current message types are TEST, TEST REPLY, CHECK and CHECK REPLY.
- OPTIONS (1 byte) : Current options are NONE and COMPRESS.
- PAYLOAD SIZE (2 bytes) : Indicates the size (in bytes) of the payload.
- PAYLOAD (variable) : N bytes make up the payload. The payload's maximum size is 64 KiB.

Every time the poller needs to query its clients a CHECK message is built and broadcasted to all clients that are managed by any monitor. When the client receives this CHECK message it proceeds to run all checks that the server instructs it to run. After all checks are executed their outputs are collected and a CHECK REPLY message is built and sent to the server.

The TEST and TEST REPLY messages are not yet implemented (just defined). The idea is to have a user-controlled way to explicitly force the run of specific checks. This is useful because if a check is not working as expected and a developer or sysadmin fixes it, then it doesn't not make sense to wait until the next poll round to verify that check performs as expected or fails again. This feature will be implemented in a next release along with a small console that allows the user to have more control of the running server.

The payload is always a JSON. The decision behind using JSON is that provides flexibility and an easy way to validate and convert data that comes from the other side of the network. Besides that it also allows the final user to layout the data field of checks as she or he wishes. This also has downsides : more bytes are sent through the network and an extra overhead is payed every time we serialize and deserialize a JSON string.

Currently messages are not being compressed at all. This feature makes sense only if the client replies a message longer than 64 KiB. This feature will be certainly included in a future release.

### 1.9.8 Class diagrams

Sometimes class diagrams help you see the big picture of a design and also act as useful documentation. Here are some diagrams that may help you to to understand what words make cumbersome to describe.

The diagrams contains the most relevant classes of both Radar server and client. Only the most important methods of every class are mentioned. You should follow these diagrams along with the code to have a detailed understanding about what's happening on a certain part of the project.

Radar client :

| RadarClient | RadarClientLauncher |
|---|---|
| | |

Radar server :

| RadarServer | Server |
|---|---|
| | |

| Monitor | ServerConfig |
|---|---|
| | |

Notes :

- RadarServerLauncher is analogous to RadarClientLauncher.

## 1.10 Limitations

Radar is a brand new project and here are some things that you should know about its current status :

- Concurrent checks : Radar does not support concurrent check execution at the moment.
- Concurrent plugin execution : At the moment all plugins are executed sequentially, this is in a way identical to the checks limitation described above.
- Passive checks : There's no passive check support yet. This feature will certainly be implemented in the near future.
- SNMP checks : SNMP is not supported. It hasn't been yet defined if this feature will be supported at all.
- SSL/TLS : Is not yet supported but certainly it's going to be included on a future release.
- IPv6 addresses are not supported yet.
- Both Windows client and server need to be improved considerably. I/O Completion Port support has been developed but is not working properly, consequently Radar relies on the inefficient select system call.
- No start/stop scripts are supplied for any platform yet.

These limitations are intended be overcomed as the project evolves. Currently Radar's status is ALPHA and proper and expected behaviour needs to be assured before moving to further features. Also minor changes can be expected at this stage.

## 1.11 License

Radar is distributed under the GNU LGPLv3

## 1.12 Acknowledgements

- To Ricardo Maia for its wonderful Radar Openclipart logo.
- To John Curley for reviewing the english version of the documentation.

## 1.13 Authors

- Lucas Liendo <liendolucas84@gmail.com>

  Drop me a line if you find this software useful or if you find something is not working properly or as you expect. Bug reporting, suggestions, missing documentation and critics (both positive and negative) of any kind are always welcome !