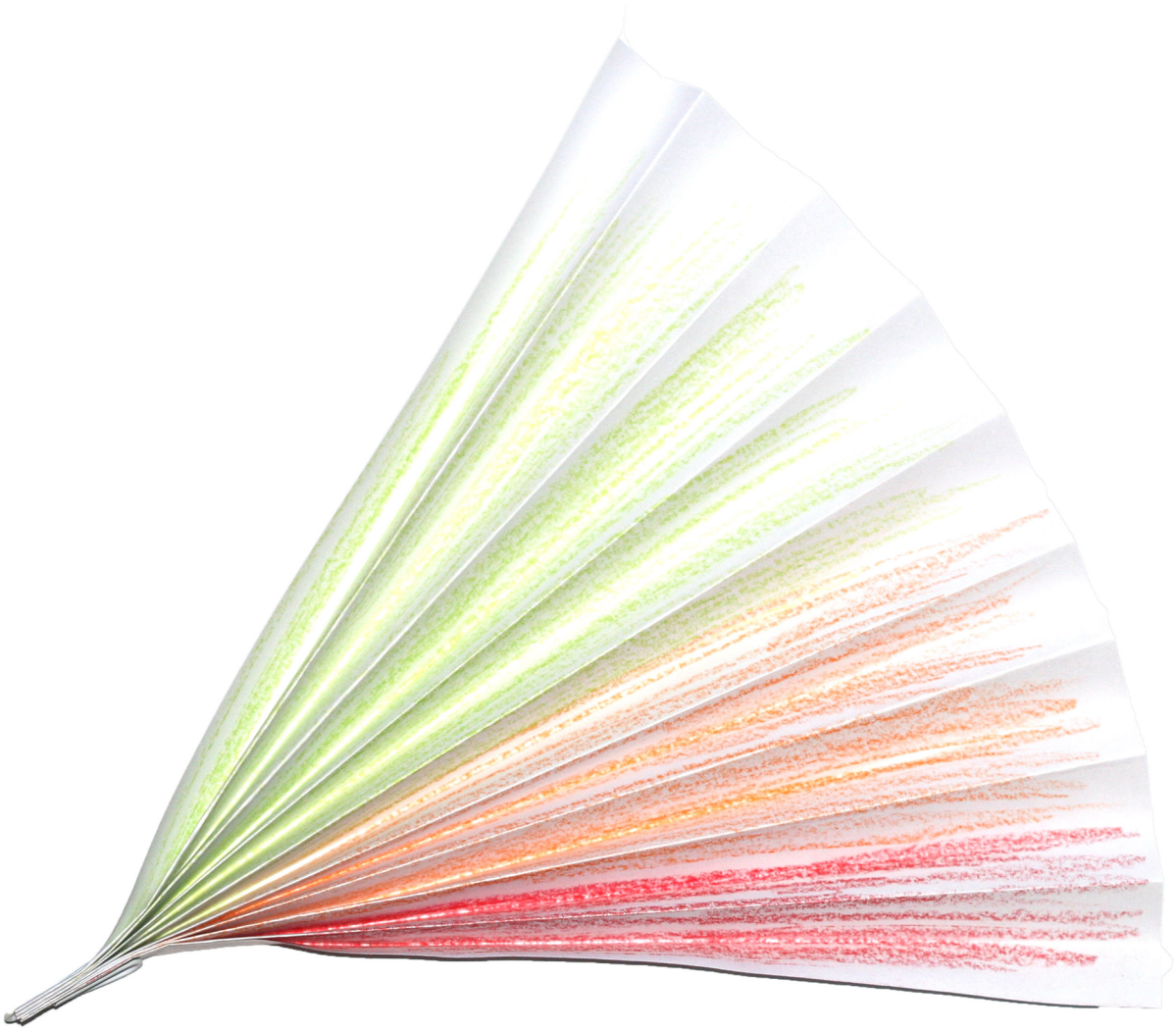
Sensu

Monitoring and Metrics



John VanDyk

Sensu

Monitoring and Metrics

John VanDyk

This book is for sale at http://leanpub.com/sensumonitoringandmetrics

This version was published on 2016-03-18

Tweet This Book!

Please help John VanDyk by spreading the word about this book on Twitter!

The suggested hashtag for this book is #sensumonitoringandmetrics.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#sensumonitoringandmetrics

Contents

SAMPLE CHAPTER

This is a sample chapter from the book Sensu: Monitoring and Metrics[1].

[1] https://leanpub.com/sensumonitoringandmetrics

CHECKS

Checks and their plugins do the grunt work of a Sensu-enabled system. They check disk space, check CPU utilization, check that outside systems can be reached…they check anything that you want them to monitor.

There are two parts to a check. The **check definition** is written in JSON and provides metadata about the check to Sensu, including which command should be run and how often.

The command itself is something that can be executed at the command line and is typically the name of a script, like a shell script named `check-mem.sh` or a Ruby script named `check-load.rb`. These scripts plug check functionality into Sensu's check definition and you will seem them referred to as **check plugins**. Think of plugins as the business end of a check. Check plugins live on the client machine, typically at `/etc/sensu/plugins`.

Check definitions describe the check. **Check plugins** are the check.

When a check plugin runs, it returns one of four possible exit codes:

Exit Code	Meaning
0	all is well
1	warning
2	critical
3	unknown

There are three kinds of checks: standalone checks, which run independently on the client and push results to Sensu Server, regular checks which are scheduled by Sensu Server but run on the client, and keepalive checks which simply alert if a client goes offline.

Standalone Checks

A standalone check is the simplest of checks. A computer running Sensu Client executes a check plugin and sends results to the results queue in RabbitMQ. Sensu Server listens to the results queue, so it picks up the results and sends them to the proper handler.

Note that no configuration for this check is necessary on Sensu Server. Standalone check definitions are stored in `/etc/sensu/conf.d` on the client. Additionally, they declare themselves standalone by using the key-value pair `"standalone": true` in the definition.

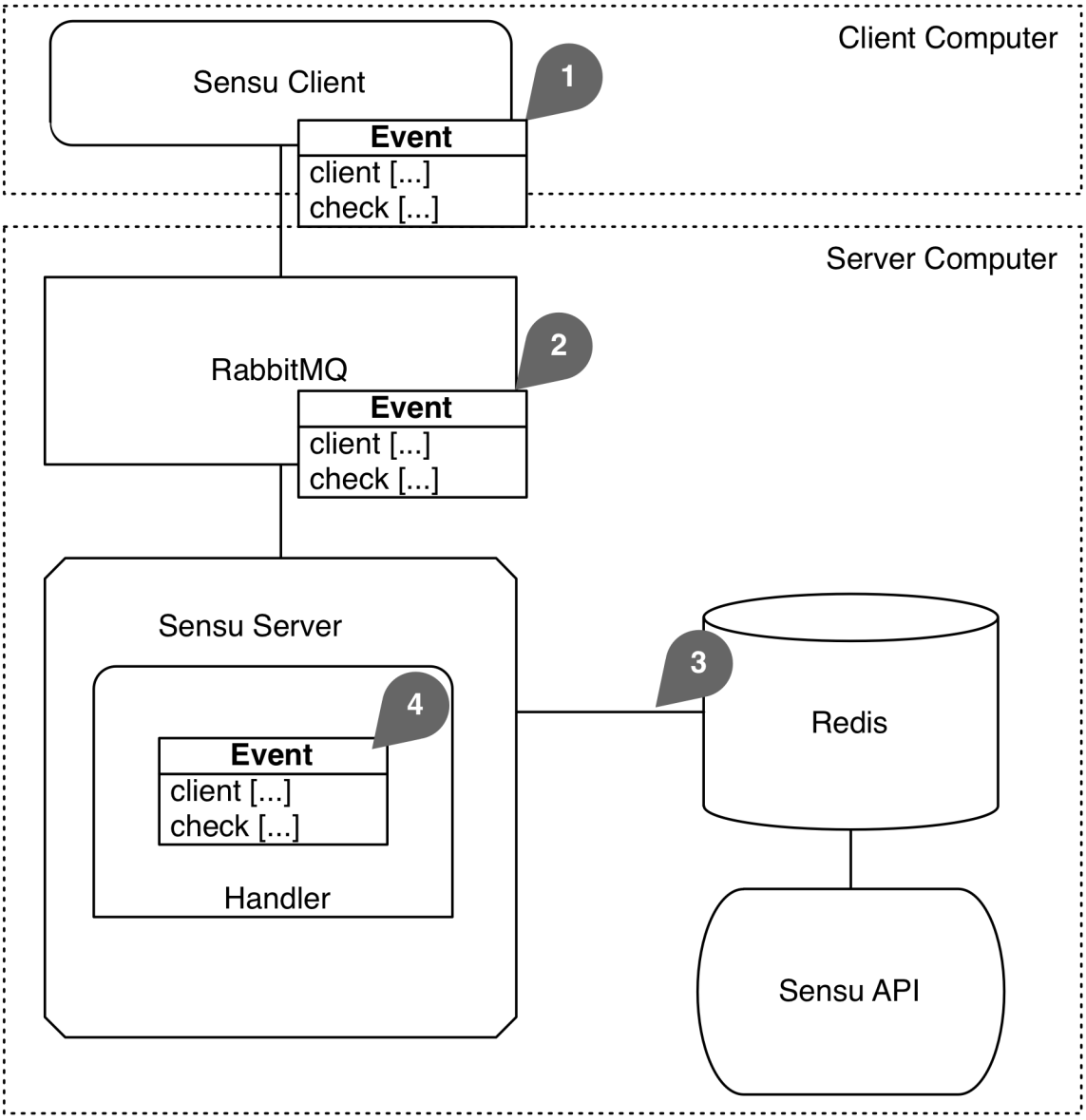


Figure 4-1. Conceptual model of standalone check. It (1) emanates from the client, (2) is sent to RabbitMQ, and (3) is picked up by Sensu Server from the RabbitMQ results queue and history stored in Redis, then (4) potential processing by a handler.

In the example below, the check definition describes a check that will run once an hour, because the `interval` key has been set to 3600 seconds.

Automatic offsets

Sensu Client schedules the execution of standalone checks with a two-second offset, so if you have 30 checks to run every hour they do not all go off simultaneously on the hour.

Example Standalone Check

Let's examine a standalone check that throws a critical error if it is a Monday, and a warning if it is a Friday. First we'll write the check definition, which would be stored at `/etc/sensu/conf.d/checks/day_check.json`:

Check definition: /etc/sensu/conf.d/checks/day_check.json

```
{
  "checks": {
    "day_check": {
      "command": "check-day.sh",
      "standalone": true,
      "interval": 3600
    }
  }
}
```

The name of the check plugin (`check-day.sh`) was given in the `command` key of the check definition. How does Sensu know where to look for `check-day.sh`? It hands this task off to the shell, which looks at the `$PATH` environment variable like any other command. The startup script for Sensu adds the plugins path (by default, `/etc/sensu/plugins`) to the `$PATH`. If you want to run a plugin that does not live in a directory defined by `$PATH` you can always provide the absolute path to the check plugin, e.g., `"command": "/opt/foo/bar.sh"`.

Here is the code of `check-day.sh`, the check plugin that will actually run.

Check plugin: /etc/sensu/plugins/check-day.sh

```
#!/bin/sh

DAY=$(date '+%A')

if [ "$DAY" == "Monday" ]; then
  echo "CRITICAL - It is Monday"
  exit 2
elif [ "$DAY" == "Friday" ]; then
  echo "WARNING - It is Friday"
  exit 1
else
  echo "OK - Just another day"
  exit 0
fi
```

Notice that as expected, the check plugin returns one of the standard exit codes.

Advantages of Standalone Checks

Standalone checks run only on the client, with no overhead on the server for scheduling and publishing check requests. Therefore standalone checks are useful for checks that you know should run on a specified schedule. For example, if you want to collect memory usage metrics from all of your servers every minute and this is never going to change, you might set up a standalone check.

Disadvantages of Standalone Checks

Because Sensu Server has no control over standalone checks (other than receiving results and giving them to handlers) any changes in standalone checks must be done on the client machines, typically using a configuration management tool like Chef or Puppet. With regular (non-standalone) checks, making a change to the Sensu Server configuration in `/etc/sensu/conf.d` (and restarting Sensu Server) is enough to start, stop, schedule, add or remove a check. This is one of Sensu's advantages: simple centralized configuration. (If you have strong configuration management this may not be much of an advantage.)

Likewise, the segregation of checks into different subscriptions (one for webservers, one for databases, one for core operating system checks, etc.) is subverted by standalone checks, since standalone checks do not use subscriptions.

It is also worth considering that Sensu Server will know immediately if a check failed if Sensu Server is the one that issued it. Sensu Server has no knowledge of failing standalone checks unless the client fails to respond to keepalive checks.

Finally, the ability to aggregate checks (see Aggregate Checks, below) where Sensu Server issues many checks across a group and alerts on the combined result is not available with standalone checks since Sensu Clients don't have the same bird's eye view that Sensu Server has.

Keepalive Checks

Sensu automatically checks whether each Sensu Client is checking in on a regular basis, so there is no need to set up ping tests for Sensu Clients.

Keepalive events are sent by Sensu Client every 20 seconds. If a client doesn't check in for 120 seconds a warning will be issued and after 180 seconds a critical alert will be issued. These are defaults and are configurable on a per-client basis by adding keepalive threshold entries to the client JSON. Here is a minimal client configuration:

/etc/sensu/conf.d/client.json

```
{
  "client": {
    "subscriptions": [
      "os"
    ],
    "address": "203.0.113.2",
    "name": "sensuclient.example.com"
  }
}
```

Overriding Default Keepalive Values

Here is one that explicitly contains the default threshold values for keepalives:

/etc/sensu/conf.d/client.json

```
{
  "client": {
    "subscriptions": [
      "os"
    ],
    "address": "203.0.113.2",
    "name": "sensuclient.example.com",
    "keepalive": {
      "thresholds": {
        "warning": 120,
        "critical": 180
      },
      "handler": "default"
    }
  }
}
```

Specifying Your Own Keepalive Handler

If you want to handle keepalives yourself you can change the handler to a different value or change the key from `handler` to `handlers` and specify a handler in addition to the default:

/etc/sensu/conf.d/client.json

```
{
  "client": {
    "subscriptions": [
      "os"
    ],
    "address": "203.0.113.2",
    "name": "sensuclient.example.com",
    "keepalive": {
      "thresholds": {
        "warning": 120,
        "critical": 180
      },
      "handlers": [ "default", "myhandler" ]
    }
  }
}
```

For example, if you are working in a cloud environment and virtual machines disappear because you want them to, you might want to have a handler that checks whether a machine has been decommissioned when a keepalive alert is received. It could check your database of live boxes and remove the client from Sensu Server's list of clients using the Sensu API.

RabbitMQ Heartbeat

RabbitMQ has its own heartbeat between AMQP clients and the RabbitMQ server which defaults to 600 seconds. The heartbeat default can be verified by issuing the following command on the RabbitMQ server:

```
rabbitmqctl environment
```

To avoid the situation in which, on a flaky network, the connection between Sensu Client and RabbitMQ is reset, but not restored until after 600 seconds, during which Sensu Server has issued critical alerts for client keepalives not coming in, the RabbiqMQ heartbeat can be set to a value less than the warning threshold value for Sensu Client. This is accomplished by specifying the heartbeat value in the RabbitMQ server configuration file at `/etc/rabbitmq/rabbitmq.config`:

/etc/rabbitmq/rabbitmq.config

```
[
  {rabbit, [
    {heartbeat,119},
    ...
  ]}
]
```

Adjusting the RabbitMQ heartbeat value should not normally be necessary on reliable networks, as it takes more resources to generate and consume heartbeats when the heartbeat is more frequent.

Regular Checks

While standalone checks run on Sensu Client, much of the value of running Sensu comes from the ability of the client to subscribe to a certain set of checks. These checks have their check definitions defined on the master Sensu Server, which periodically publishes a check definition to a RabbitMQ queue. Sensu Clients listening to that queue receive the check definition, run the check plugin code that the check requires, and return results to the RabbitMQ results queue.

Ask yourself the following questions when writing a check definition:

Is the command (that is, the check plugin) I will be asking the clients to run available on the clients? Typically these are scripts that live in `/etc/sensu/plugins` on the client. You can write your own, use scripts from the Sensu community plugins repository[2], or use Nagios plugins.

Which set of servers to I want to run the check on? This determines the values in the subscribers section of the check definition. The queues you list here are arbitrary; you can make up whichever groups you wish. You just have to be sure that your Sensu Clients are configured to listen to those queues; they must be listed in the `subscriptions` section of `/etc/sensu/conf.d/client.json` on the clients.

What should happen with the results? If warnings or critical alerts are received in response to this check, what should happen? An email? Pager? Self-destruct sequence? The answer to this question will determine your values for the `handlers` section of the check definition.

[2] https://github.com/sensu/sensu-community-plugins

How often should this check be published to the clients? This value, in seconds, is set using the `interval` section of the check definition.

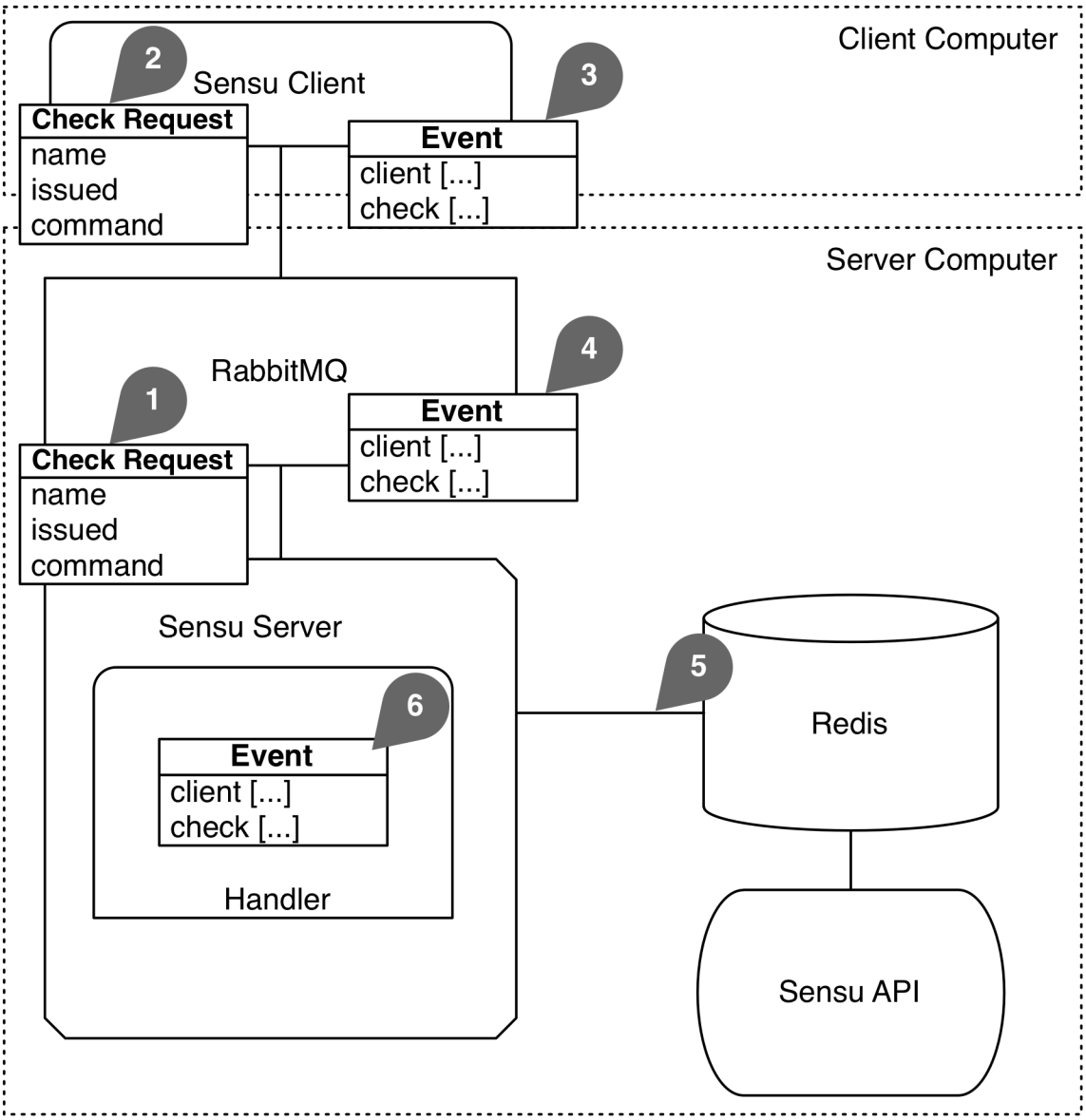


Figure 4-2. Conceptual model of regular check. It (1) is scheduled by Sensu Server and when it is time for the check it is published to RabbitMQ, (2) it is picked up by clients subscribing to that RabbitMQ queue, (3) check is run on the client and the result is sent to RabbitMQ, (4) is picked up by Sensu Server from the RabbitMQ results queue, (5) history stored in Redis, then (6) potential processing by a handler.

Below is an example of a check definition that will issue a warning if more than 90% of RAM is used and a critical alert if more than 95% of RAM is used. The check is published to all webservers:

/etc/sensu/conf.d/checks/free_ram.json

```
{
  "checks": {
    "free_ram": {
      "command": "check-ram.rb",
    "subscribers": [
      "webservers"
    ],
    "handlers": [
      "default"
    ],
    "interval": 60
    }
  }
}
```

Command Token Substitution: Overriding Check Parameters

When a check runs, it is common to want Sensu to alert when something has crossed a certain threshold. But what about the situation where two of your servers are special and you want them to alert on a different threshold than the others? Sensu provides **command token substitution** for this scenario. The check definition provides default values but the client can override them.

Consider the following check for free RAM (using the `system/check-ram.rb` script from the Sensu Community Plugins repository):

/etc/sensu/conf.d/checks/free_ram.json

```
{
  "checks": {
    "free_ram": {
      "command": "check-ram.rb",
      "subscribers": [ "os" ],
      "interval": 60
    }
  }
}
```

Investigation of the check plugin script `check-ram.rb` reveals that it has internal default thresholds which can be stated explicitly. In other words, the following check definition is equivalent:

/etc/sensu/conf.d/checks/free_ram.json

```
{
  "checks": {
    "free_ram": {
      "command": "check-ram.rb -w 10 -c 5",
      "subscribers": [ "os" ],
      "interval": 60
    }
  }
}
```

The `-w` parameter is the warning threshold and the `-c` parameter is the critical threshold. Sensu will issue a warning when less than 10 percent of RAM is left and a critical alert when less than 5 percent of RAM is left.

In order to make this overridable by the client, a different syntax is used. It consists of three colons, a key that corresponds with something in the client's `client.json` definition, and a default value to use if the key is not found on the client. So the following syntax runs the same check as before but gives us the flexibility of overriding values on the client, though it does not require it:

/etc/sensu/conf.d/checks/free_ram.json

```
{
  "checks": {
    "free_ram": {
    "command": "check-ram.rb -w :::params.ram.warning|10::: -c :::params.ram.cri\
tical|5:::",
    "subscribers": [ "os" ],
    "interval": 60
    }
  }
}
```

To override values in check definitions, add keys to the JSON that defines the client (typically found at `/etc/sensu/conf.d/client.json`). These keys must match keys in the check definition:

/etc/sensu/conf.d/client.json

```
{
  "client": {
    "name": "sensuclient.example.com",
    "subscriptions": [
      "os", "webservers"
    ],
    "address": "203.0.113.2",
    "params": {
      "ram": {
        "warning": 20,
        "critical": 10
      }
    }
  }
}
```

INested values in client.json can be accessed via dot-delimited names in check definitions.

When Sensu Client executes the check plugin, it will look for `params.ram.warning` and find our value of 20 and use that instead of the default; the same goes for `params.ram.critical`. The actual plugin check that will be executed is

```
check-ram.rb -w 20 -c 10
```

In keeping with Sensu's goal of being malleable, token substitution is entirely arbitrary. The key `params.ram.warning` was chosen because it is obvious, not because it is syntactically required. You are free to call your own token `zebra.zipzip.yeeha` if you like.

For that matter, you could do entire command substitution. Consider this definition:

/etc/sensu/conf.d/checks/free_ram.json

```
{
  "checks": {
    "free_ram": {
    "command": ":::ramcheck|check-ram.rb:::",
    "subscribers": [ "os" ],
    "interval": 60
    }
  }
}
```

The following key defined on the client will run an entirely different check plugin!

/etc/sensu/conf.d/client.json

```
{
  "client": {
    "name": "sensuclient.example.com",
    "subscriptions": [
      "os", "webservers"
    ],
    "address": "203.0.113.2",
    "ramcheck": "other-ram-check.sh"
  }
}
```

Redaction of Passwords and Sensitive Information

Redaction is the removal or suppression of sensitive information. It would be a bad thing for passwords to show up in log files, so Sensu tries not to do this.

Built-In Redaction Keys

Sensu Client will automatically redact values from JSON keys before sending JSON events across the wire. The following keys will be redacted automatically:

Keys that Sensu will redact automatically

```
access_key
api_key
api_token
pass
passwd
password
private_key
secret
secret_key
```

Adding Custom Redaction Keys

Additional keys to be redacted can be entered in your `client.json` file. For example, the following will additionally redact any key named `secret_banana` or `passcode`:

Adding custom redaction keys to /etc/sensu/conf.d/client.json

```
{
  "client": {
    "name": "sensuclient.example.com",
    "subscriptions": [
      "os", "webservers"
    ],
    "address": "203.0.113.2",
    "redact": [
      "secret_banana",
      "passcode"
    ]
  }
}
```

Redacted values will appear as the string `REDACTED` in logs.

Safe Mode

Safe mode is enabled by setting the `safe_mode` Boolean key in the Sensu Client configuration.

/etc/sensu/conf.d/client.json

```
{
  "client": {
    "name": "sensuclient.example.com",
    "safe_mode" : true,
    "subscriptions": [
      "os", "webservers"
    ],
    "address": "203.0.113.2",
  }
}
```

If the `safe_mode key` is absent or set to `false`, checks proceed normally. But if `safe_mode` is set to `true`, Sensu Client will require that the check definition exists *on the client as well as on the server*. If it does not, Sensu will return a status of `3` (unknown) for the check and output of `"Check is not locally defined (safe mode)."`

Safe mode? Safe from whom?

This is safe mode from the client's perspective: it is safe from a rogue Sensu Server issuing a check definition with something really nasty in the command. It essentially tells Sensu Server "I will not run your checks blindly; if you publish checks to me I will only run the checks that I know about here in my local configuration."

Note that for a client and server running on the same box, both using `/etc/sensu` for their configuration files, setting the `safe_mode` key on the client will have no effect. That is because Sensu Server and Sensu Client both read the check definitions so the checks exist both in the client and server configurations and therefore the requirements of safe mode are met.

This key is not used for standalone checks, as it doesn't make any sense: `safe_mode` is used to enforce that check definitions exist on the client, and standalone checks by their very nature exist on the client.

Aggregate Checks

Sometimes you want Sensu to send out one check but receive a whole lot of results. For example, you might have 50 database servers and want to know how many of them are running at high load. If more than 90 percent of them report high load, you probably want to spin up more servers. Here's an example of the check definitions you would write in that situation. First, a check is needed to check the load on the databases:

/etc/sensu/conf.d/checks/load_check.json

```
{
  "checks": {
    "load_check": {
      "command": "check-load.rb -w 10,20,30 -c 25, 50, 75",
      "subscribers": [ "databases" ],
      "interval": 60,
      "aggregate": true,
      "handle": false
    }
  }
}
```

Note that `handle` is set to `false`, so no handler will be alerted when the check results come back. Also, the `aggregate` key has been set to `true` which tells Sensu Server to persist the check results into Redis for inspection later.

Standalone checks do not work with aggregate checks. In this example, the request for the client to run the check-load.rb check plugin must emanate from Sensu Server. Why? Because the results from these checks will be aggregated in a single place in Redis, stored under a compound key consisting of the check name and the timestamp when the check request was issued by Sensu Server (see aggregate_check_result() in process.rb if you are curious). If the checks were standalone checks, each client would send results with differing timestamps.

OK, so check results have now been stored in Redis for inspection later. When is "later"? It's when a second check runs to evaluate the aggregate results stored in Redis. This check is going to use the `sensu/check-aggregate.rb` script from the Sensu Community Plugins repository.

/etc/sensu/cond.d/checks/aggregate_load_check.json

```
{
  "checks": {
    "aggregate_load_check": {
      "command": "check-aggregate.rb -c load_check -W 75 -C 90",
      "subscribers": [ "decisionmaker" ],
      "interval": 300,
      "handlers": [
        "irc", "database_spinner_upper"
      ]
    }
  }
}
```

This check will be run by those listening to the `decisionmaker` queue, presumably just the master Sensu Server. In reality, any Sensu Server that can reach the Sensu API can do it. Or for that matter a Sensu Client that can reach the Sensu API and is subscribed to the `decisionmaker` queue could receive and run this check, though only one should since we want only a single check of aggregate data.

The command being run is

```
check-aggregate.rb -c load_check -W 75 -C 90
```

This tells the `check-aggregate.rb` script to inquire about the aggregate results stored in Redis. It will do this by asking the Sensu API for the results.

Aggregate Threshold Values

The warning and critical threshold values have very special meaning with aggregate checks. A warning value of 75 means that if more than 75% of clients that responded to the `load_check` check

definition Sensu Server published had non-zero exit values (that is, they issued a warning or critical alert due to high load), the `aggregate_load_check` will return an exit code of `1` (warning).

If more than 90 percent of clients running `load_check` have a non-zero exit code, `aggregate_load_check` will return an exit code of `2` (critical).

If the `aggregate_load_check` has a nonzero exit code, it will have its results sent to its handlers, just like any other Sensu check. In the example above, Sensu is configured to send results to the `irc` handler so that an appropriate message is posted to an IRC channel, and it also fires off a theoretical `database_spinner_upper` handler so that more database servers can be started.

Cleaning Up Aggregate Data

With many aggregate checks, you may be worried that the amount of data stored in Redis will grow and grow. Have no fear. Every 20 seconds the master Sensu Server prunes aggregate check results from Redis, leaving the 20 most recent check results. Just before it embarks on this task it writes "`pruning aggregations`" to the sensu-server.log file.

Self-Documenting Checks

Because arbitrary key-value pairs can be created in your check definitions and these values are passed along to the handlers, you can include hints in your checks that your junior sysadmins can use when responding to an alert. For example:

/etc/sensu/conf.d/checks/legacy_database_check.json

```
{
  checks": {
    "legacy_database_check": {
      "command": "check_foxpro.rb",
      "handler": "mailer",
      "subscribers": [ "antiqueservers" ],
      "interval": 60,
      "playbook": "Before restarting the legacy database server, make sure no on\
e is using the microwave or you will trip the circuit breaker."
    }
  }
}
```

Note that in the above check the `playbook` key is specific to the `mailer` handler. The point here is that you can define additional information in the check definition and that the handler will have access to that information. Your handlers may need to be customized to access the information.

Running Checks as a Privileged User

When Sensu Client is running a check and needs to run as a privileged user, the sensu system user might need additional privileges assigned in `/etc/sudoers.d`.

/etc/sudoers.d/sensu

```
Defaults:sensu !requiretty
Defaults:sensu secure_path = /usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin\
:/usr/bin

sensu ALL = NOPASSWD: /etc/sensu/plugins/mycheck.rb
```