

# RAVE Administrator's Guide

Phil Groce  
CERT Network Situational Awareness Group

June 27, 2007

## Contents

<b>I</b>	<b>Introduction</b>	<b>2</b>
1	About this document	2
2	Additional Information	2
<b>II</b>	<b>Installing RAVE</b>	<b>2</b>
3	Requirements	3
4	How RAVE is distributed	3
4.1	Distutils package . . . . .	3
4.2	Raw archive . . . . .	3
5	RAVE Components	4
5.1	RAVE core and analysis libraries . . . . .	4
5.2	RAVE "Extras" . . . . .	4
5.2.1	<i>raved</i> configuration files . . . . .	4
5.3	Sample binaries . . . . .	4
5.4	Documentation . . . . .	4
<b>III</b>	<b>Writing RAVE Analyses</b>	<b>4</b>
6	RAVE Operations	5
7	Using operations within operations	6
8	File Operations	8
9	Passing options to operations	8
10	Managing data retention with <i>@expires</i>	9
11	Operation versioning	9
12	Typemaps	10

13 Specifying output type	12
<b>IV <i>raved</i> and <i>mod_rave</i></b>	<b>12</b>
14 <i>raved</i> and <i>mod_rave</i> Configuration	12
14.1 System Requirements . . . . .	12
14.2 <i>raved</i> Configuration . . . . .	13
14.2.1 Command-line configuration options . . . . .	13
14.2.2 Namespace configuration . . . . .	14
14.2.3 Log configuration . . . . .	14
14.3 <i>mod_rave</i> configuration . . . . .	14
15 Writing analyses for <i>raved</i>	15
15.1 Namespaces . . . . .	15
15.2 Internal namespace organization . . . . .	16
15.3 Exporting Operations . . . . .	16
<b>V Appendices</b>	<b>17</b>
A List of expiration strategies	17
B List of RAVE type specifiers	18

## Part I

# Introduction

The Retrospective Analysis and Visualization Environment (RAVE) is a framework for writing and sharing code that generates data analysis and visualization products. The goal of the RAVE project is to provide an environment in which to write analyses that run faster and integrate smoothly into a number of operating environments.

RAVE minimizes the cost of executing expensive analyses by caching their results. For most functions, it simply caches the return value of the function; when the same function is called again with the same arguments, it returns the cached value rather than running the function again. (This process is called *memoization*.) If the analysis returns a large amount of data, the analysis can write the result directly to a file, as well.

Beyond caching, RAVE provides interfaces for sharing both the analyses themselves as services, and for sharing the results cache directly among multiple users.

At its core, RAVE is a set of Python libraries. The simplest way to use RAVE is at the Python command prompt.<sup>1</sup> One may also write applications to run analyses in a different user interface, or to interact with other applications across a network. One such application, *raved*, comes with RAVE. *raved* provides RAVE analyses via HTTP.

RAVE was written to support the analysis tasks of network operations centers performing queries on large sets of network monitoring data. Although this continues to be the focus of development, RAVE is designed simply to improve performance of expensive processes with caching, a desirable feature in many environments.

## 1 About this document

This guide takes you through the installation of RAVE and shows you how to RAVE-enable your code. We also discuss how to install and configure *raved*, and how to use some existing RAVE-enabled libraries for querying databases and other large datasets, notably network flow data stored using SiLK.

## 2 Additional Information

There is additional documentation on specific functions and classes available via Pydoc, the Python inline code documentation system. To view this documentation at the command-line, type `pydoc modulename`, where *modulename* is a RAVE library module, such as `rave.plugins.decorators`.

---

<sup>1</sup>We recommend iPython (<http://ipython.scipy.org/>) as a full-featured interactive Python environment.

## Part II

# Installing RAVE

### 3 Requirements

RAVE requires Python, at least version 2.4. Subsequent versions may work, but are untested. Additionally, analysis operations may require additional libraries in order to work.

### 4 How RAVE is distributed

RAVE is distributed as a Python distutils-based package and as a raw source archive.

#### 4.1 Distutils package

For most systems a Python package is available, based on distutils. Python packages are distributed as gzipped TAR archives. The distutils-based package is named `rave-1.9.0.tar.gz`.

To install the python package, extract the archive and run the package's `setup.py` script. For instance:

```
tar xvzf python-1.9.0.tar.gz
cd python-1.9.0
python ./setup.py install
```

You can call `setup.py` with a number of options to customize your installation. For more information, see <http://docs.python.org/inst>.

The distutils package does not contain the `mod_rave` Apache module, documentation or examples. To obtain these, either download the unpackaged archive or the `rave-extras-1.9.0.tar.gz` archive. The contents of `rave-extras` is outlined in Section 5.

#### 4.2 Raw archive

If it is unsuitable to install from a Python distutils package, RAVE is also available as a plain gzipped TAR archive.

To distinguish between this and the Python distutils package, this file is named `rave-unpackaged-1.9.0.tar.gz`.

To install from the raw source files, extract the archive into some temporary space. This will create a directory called `rave-1.9.0`. You may then move the files to their final destination, as appropriate to your system.

For more information on the contents of this archive, see Section 5.

## 5 RAVE Components

### 5.1 RAVE core and analysis libraries

These will be installed in your Python installation's `site-packages` directory. In the unpackaged archive, they reside in the `lib/python` directory. These libraries must be in the Python module search path for RAVE to work. See <http://docs.python.org/tut/node8.html> for details.

### 5.2 RAVE “Extras”

The following items are included in the unpackaged TAR archive and in the `rave-extras` archive. They are not part of the `distutils` package. Filenames are relative to the root of the archive.

#### 5.2.1 *raved* configuration files

The `doc/examples` directory contains example log and namespace configuration files for *raved*. Also included is a sample init script for managing *raved* as a system service.

### 5.3 Sample binaries

The `bin/samples` directory contains sample binaries illustrating how to write applications that talk to RAVE. Currently, the only application in this directory is `rave-get.py`, a command-line application which requests analyses from *raved*.

### 5.4 Documentation

The `doc` directory contains documentation, including this document.

## Part III

# Writing RAVE Analyses

In order for RAVE to do its job, needs to know what results it should cache, how the results are returned, and other meta-information about the results, like how long they can be cached before they must be recomputed.

In Python, we can obtain this information fairly unobtrusively through function decorators.<sup>2</sup> Decorators can also change the behavior of functions; RAVE uses this to implement its results caching. RAVE analyses, then, are simply Python functions with some special decorators attached.

---

<sup>2</sup>For more information on Python decorator syntax, see <http://www.python.org/dev/peps/pep-0318/> and <http://docs.python.org/ref/function.html>

## 6 RAVE Operations

To begin, let's turn a normal Python function into a RAVE analysis.

```
def complex():
    # Do a big complex query
    return query_results
```

To enable RAVE to cache the output of this function requires two additional lines of code:

```
from rave.plugins.decorators import op
@op
def complex():
    # Do a big complex query
    return query_results
```

We import some symbols from *rave.plugins.decorators* and decorate *complex* with *@op*. This signifies that *complex* is a RAVE *operation*. Operations are identical to normal Python functions, but RAVE stores their return value in its cache; if someone runs the function again with the same arguments, RAVE returns the cached value instead of running the function again. We can make this obvious (with a different example) at the command-line:

```
>>> from rave.plugins.decorators import op
>>>
>>> @op
... def add(a, b):
...     print "Executing add"
...     return a + b
...
>>> add(1, 2)
Executing add
3
>>> add(1, 2)
3
>>> add(1, 3) # different args, should run function
Executing add
4
```

Here's another example. The following function returns the *n*th number in the Fibonacci sequence:

```
>>> @op
... def fib(n):
...     if n <= 0:
...         return 0
...     elif n == 1:
...         return 1
...     else:
...         return fib(n-2) + fib(n-1)
```

Because *fib* is an *@op*, it caches its output. Because it's defined recursively, it implicitly caches all its intermediate work as it goes. You can see the benefits using Python's *timeit* module:

```
>>> from timeit import Timer
>>> Timer('fib(50)', 'from test import fib').timeit(1)
0.41100001335144043
>>> Timer('fib(50)', 'from test import fib').timeit(1)
0.0014078617095947266
>>> Timer('fib(25)', 'from test import fib').timeit(1)
0.0012981891632080078
```

The times above are in fractions of a second. In other words, all those calls took under a second. By comparison, here's how long it took on one machine to make similar calls on a *fib* function that was not an *@op*:

```
>>> Timer('fib(20)', 'from test import fib').timeit(1)
0.014575958251953125
>>> Timer('fib(30)', 'from test import fib').timeit(1)
1.2909920215606689
>>> Timer('fib(40)', 'from test import fib').timeit(1)
160.64894795417786
```

## 7 Using operations within operations

You can use RAVE operations within other operations. In fact, this is a very effective way to use cached data. Consider the following:

```
from rave.plugins.decorators import *

def aggregate(data):
    # aggregate data points in *data into
    # a single report
    return aggregated

def visualize(proto, port):
    # generate an amalgamated visualization
    # of proto and port data
    return visualized

@op
def generate_hourly_port_report(hour, port):
    # generate the report
    return generated

@op
def generate_hourly_proto_report(hour, proto):
    # generate the report
    return generated
```

```

@op
def generate_proto_reports(day, proto):
    hourlies = []
    for hour in day.hours():
        hourlies.append(
            generate_hourly_proto_report(hour, proto)
        )
    return aggregate(hourlies)

@op
def generate_port_reports(day, port):
    hourlies = []
    for hour in day.hours():
        hourlies.append(
            generate_hourly_port_report(hour, port)
        )
    return aggregate(hourlies)

#
# Putting it all together...
#

@op
def generate_daily_reports(day):
    daily_protos = []
    for proto in (1, 6, 17):
        daily_protos.append(
            generate_daily_proto_report(day, proto)
        )
    proto_reports = aggregate(daily_protos)

    daily_ports = []
    for port in (22, 25, 80):
        daily_ports.append(
            generate_daily_port_report(day, port)
        )
    port_reports = aggregate(daily_ports)
    return visualize(daily_protos, daily_ports)

```

The function *generate\_daily\_reports* runs reports for three IP protocols and three TCP ports. To get its data, it calls *generate\_port\_report* and *generate\_proto\_report* for each protocol or port; they, in turn, return an aggregate of the hourly reports for each protocol or port.<sup>3</sup>

If we run *generate\_daily\_reports*, say, in a cron job at the end of the day (for looking at the next morning), we cache not only the single daily report visualization, but all the data used to generate it, broken down by port or protocol and by hour. Drilling into the data used to generate the report will be much

---

<sup>3</sup>Some readers might notice that *aggregate* is not an *@op*. In this example, the results of a specific call to *aggregate* are only of interest to that specific caller; we don't expect someone to want the results of aggregating this particular set of daily reports again, unless they're calling *generate\_daily\_reports*, which is already cached. Another reason not to make a function an *@op* is when the process of retrieving a cached result is no faster than computing the result. The *add* example used previously is such a case—in the real world, it takes much less overhead to add two numbers than to retrieve the results from an on-disk cache.



easier because the data was implicitly cached by *generate\_daily\_reports*. Alternately, we could run the *generate\_hourly\_report* scripts after each hour, and the end-of-day report will run much more quickly.

## 8 File Operations

Some things are inconvenient to pass as a function return value, but nice to cache. Large data sets are inconvenient to store in memory. As a result, tools that manipulate these sorts of data frequently work in terms of files on disk. Visualizations often fall into this category, too—most APIs for generating or manipulating images expect to work with files on disk.

RAVE supports this way of working with the *file operation*. Writing a file operation is very similar to writing a normal operation. Consider this example:

```
import os
@op_file
def thumbnail(out_file, in_file, width, height):
    os.system(
        "convert %s -size %d %d %s" % (
            in_file, width, height, out_file))
```

This function generates a thumbnail from (presumably) the name of a larger image on the disk using the *convert* utility from ImageMagick.<sup>4</sup> *out\_file* points to a file which the RAVE cache will manage.

It is decorated with *@op\_file* rather than *@op*. The behavior RAVE expects from an *@op\_file*-decorated function is somewhat different from that of an operation. Most importantly, its output is expected to be written to the file whose name is passed via the first argument; as a result, its return value is ignored.

Note, too, that RAVE supplies the first argument to the function. When calling the function, therefore, omit the first argument:

```
new_file = thumbnail("/path/to/input/file.png", 100, 100)
```

Finally, RAVE passes the filename as the return value of the function, even though the function returns nothing (and its return value would be ignored if it did).

## 9 Passing options to operations

You can control how an *@op* or an *@op\_file* runs by passing it special arguments that RAVE reads before executing the underlying function. To distinguish these from normal arguments, we call them *options*.

For example, what if we want to use *thumbnail* as a normal function? Perhaps we want to use it as an intermediate step in the production of a larger image, and we don't care to cache the created file because it will never be used anywhere else. To do this, pass the operation the *rave\_to\_file* option:

```
# Write thumbnail to a temporary file
new_file = thumbnail("/path/to/input/file.png", 100, 100,
    rave_to_file="mythumb.png")
```

---

<sup>4</sup><http://www.imagemagick.org>

When called like this, *thumbnail* will write its output to the file “myfile.png”. (It will also return this file-name, so in this example, *new\_file* will contain the string `'myfile.png'`. This will always be identical to the name in *rave\_tofile*, and may therefore be safely ignored.)

RAVE operations support the following options:

***rave\_tofile*** Value: File name. Execute the *@op\_file*, writing the output to the specified file.

***rave\_refresh*** Value: *True* or *False* (default). Assume any cached value for this operation has expired and should be refreshed.

***rave\_deep\_refresh*** Value: *True* or *False* (default). Like *rave\_refresh*, but also refresh any operations this operation may call.

***rave\_dry\_run*** Value: *True* or *False* (default). If a cached version exists, return it; otherwise, return *None*.

## 10 Managing data retention with *@expires*

The most common reason to remove data from a cache is to reclaim space the cache occupies. However, even assuming infinite disk space, some results will require recomputation after some period of time. When dealing with very recent time-series data, for instance, it may be that new data for the time period in question will arrive shortly. It may improve performance to cache the results briefly, but soon the cached data will be too outdated to continue using.

Use the *@expires* decorator to control how long to retain cached data. *@expires* takes as its input a *cache expiration strategy*, which is a function whose return value is a time. After that time, the data being cached will be considered invalid. For example, *soon* is an expiration strategy for operations whose values should not stay in the cache very long:

```
import time
def soon(req):
    return time.time() + 300 # five minutes

@op
@expires(soon)
def volatile_data(x, y, z):
    # ...
```

Expiration strategies like *soon* take one argument, a Request object.<sup>5</sup> They return a Unix time value after which the results returned should be considered invalid.

See Appendix A for further discussion of the expiration strategies that come with RAVE.

## 11 Operation versioning

Something else can invalidate the results of an operation besides age—changing the operation itself. When RAVE is determining whether it has cached data for an operation call, it takes four things into account: the

---

<sup>5</sup>The Request object is documented in detail in the *rave.plugins.decorators* module. For more information, type *pydoc rave.plugins.decorators* at the command line.

operation itself, the arguments to the operation, the age of the data, and whether the operation has changed since the data was cached.

By default, RAVE uses the modification time of the file in which the operation was defined to determine if the operation was changed. This works most of the time; however, there are times it might cause problems. For example, perhaps there was only a small cosmetic change to a function (like renaming a variable), and want to continue returning cached data. Perhaps the operation in question didn't change at all, only something else in the same file.

If it is important to only invalidate the cache data when the operation substantively changes, consider giving the operation a specific version. Beyond expiration, RAVE will only invalidate the cached results of a versioned operation if the version changes, regardless of whether the file containing it changed.

Versioning an operation is fairly simple; just add the `@version` decorator:

```
@op
@expires(duration_strategy(300))
@version("20070211")
def my_operation(x, y, z):
    # ...
```

This example uses the string "20070211" as the version identifier, but it can be any valid Python value. (We recommend encoding the date of change in the version number, in order to know when it was last changed.) Whenever it changes, the cache results from that operation are no longer valid. Similarly, if the file or operation changes, will remain valid unless the version identifier has changed.

In environments with large amounts of cached data, versioning can provide a way to preserve cached data that is still useful even though an operation or module may have changed. When versioning an operation, however, take care to change the version every time the operation changes in a meaningful way; otherwise, the cache may inadvertently store inaccurate data.

## 12 Typemaps

Most of the time, `@op`, `@op_file`, `@expires` and `@version` are sufficient to make the most out of RAVE. You could use the following operation with no problems for quite some time:

```
@op
@expires(duration_strategy('30m'))
def top_n(n):
    "Get the top n elements from data."
    rows = fetch_sorted_data()
    return [rows[x] for x in len(rows) if x < n]
```

However, this code has two limitations, which, depending on its purpose, may or may not be significant:

1. This function generates the same results whether its first argument is `10` (an integer) or `10.0` (a floating-point value). However, to the cache, they are different arguments, which may result in needless cache misses.
2. All operations exported to `raved` initially receive their input as strings. This operation will be called as `top_n("10")`, which will produce an error.

The fundamental problem is that Python doesn't know what type *n* is. In Python, this is usually no problem; it implicitly converts what it can when something is used, and raises an exception when it can't.

Since we are using the arguments to the function as identifiers in a cache, however, equivalent types with different representations will cause them to be stored in different parts of the cache, so RAVE may not notice cache data for an operation invocation, even if it is still valid. This is also a problem in service-oriented architectures which need to know the arguments the function expects so it can convert them before calling the function.

To get around this limitation, RAVE has the concept of *typemaps*. RAVE analysis developers can use typemaps to let the cache and service applications know what argument types an operation expects. As a bonus, typemapped operations will already have their arguments converted to a standard representation—a very handy feature for data types with multiple valid representations, such as dates.

Add a typemap to an *@op* or an *@op\_file* using a decorator:

```
@op
@typemap(str, int, iso_date, iso_date)
@expires(duration_strategy('2h'))
def my_op(name, num, sdate, edate):
    # ...
```

This typemap indicates that the argument *name* is a string, *num* is an integer, and *sdate* and *edate* are variables of the type *iso\_date*.

You can specify a typemap in a number of ways—the syntax is essentially the same as a Python function call. Therefore, all of the following are equivalent:

```
@typemap(str, int, iso_date, iso_date)
@typemap(name=str, num=int, sdate=iso_date, edate=iso_date)
# order doesn't matter when all types are passed in with keywords
@typemap(sdate=iso_date, edate=iso_date, num=int, name=str)
# It is legal to use keywords for some arguments and not for others
# (be sure the non-keyword arguments are in the right position)
@typemap(str, int, sdate=iso_date, edate=iso_date)
```

The typemap performs type *conversion*, not merely type *checking*. Therefore, if the caller passes the string "1" as the first argument to *my\_op*, *name* will get the value 1—the string has been converted into an int.

The valid values for *@typemap* arguments are:

- any of Python's built-in types—e.g., *str*, *int*, *float*, *list*, *dict*. For a complete list, see <http://www.python.org/doc/2.4.4/lib/types.html>.
- one of the compound types specified in *rave.plugins.decorators* (e.g., *list\_of*, *one\_of*, *any*) See Appendix B for a fuller description of these type specifiers.
- The value *None*. This means that the only acceptable input for this parameter is the value *None*. (This is useful in some compound type converters—for instance, *one\_of(None, str)* means that a value can be a string or *None*.)
- A function which takes a single value for its input and returns the value converted into the target type. If the function cannot convert the value, it raises *TypeError*.

Do *not* specify the type of the first argument to an *@op\_file* (the output file name). RAVE will handle all aspects of this argument.

## 13 Specifying output type

Typemaps convert input into something the operation expects; to let external functions know what kind of output to expect, we provide *@mime\_type*.

As with *@typemap*, the *@mime\_type* decorator is more useful if when exporting an operation to others in a network service environment. For using operations within a Python program (including the interactive prompt), the *@mime\_type* decorator isn't necessarily very valuable.

To use MIME type, decorate an *@op* or *@op\_file* with *@mime\_type*, supplying the MIME type as a string:

```
@op_file
@typemap(int, iso_date, iso_date)
@mime_type('image/png')
@expires(duration_strategy('2h'))
def visualize_protocols(out_file, proto, sdate, edate):
    #....
```

If a MIME type is not supplied, it is assumed to be *application/octet-stream*.

## Part IV

# *raved* and *mod\_rave*

The *raved* application provides access to RAVE analyses via HTTP. In conjunction with *mod\_rave* and the Apache web server, it can be a useful visualization and analysis backend for web applications such as portals or operations consoles.

*raved* does not deliver analysis products directly to the end-user. Rather, *raved* responds to requests for different analysis products with references to the location of the product (typically HTTP URLs). The client must then retrieve the analysis product separately. This allows *raved* to focus on generating analysis products, leaving the task of serving the data to more appropriate applications, such as Apache.

*mod\_rave* is an Apache module which provides a more intuitive interface to end users. It proxies connections from clients, requests analyses from RAVE, and returns the analysis results directly in a single request.

## 14 *raved* and *mod\_rave* Configuration

### 14.1 System Requirements

*raved* comes in the main RAVE distribution, and has no additional requirements. *mod\_rave* requires Apache version 2 or later and *mod\_python* version 3.1.3 or later.

## 14.2 *raved* Configuration

*raved* uses configuration files to track its namespaces and logging. All other configuration is done using command-line switches.

### 14.2.1 Command-line configuration options

*raved* takes the following options on the command line.

- listen-addr** *raved* will listen for connections on the interface bound to this network address.
- listen-port** *raved* will listen for connections to this port.
- data-dir** Directory in which to cache intermediate analysis data.
- data-expire-after** If an analysis has no other caching strategy, expire it after it exceeds this age, in seconds. This option applies only analysis *data*. For visualizations, use **--vis-expire-after**
- data-purge-every** *raved* will periodically remove expired items from the data cache to save disk space. This option controls how often this happens. If unspecified, *raved* will purge the data cache every two hours.
- output-dir** Directory where *raved* should write visualizations.
- vis-expire-after** Similar to **--data-expire-after**, this parameter determines when existing visualizations will no longer be served in response to requests and may be cleaned up.
- vis-purge-every** *raved* will periodically remove expired items from the visualization cache to save disk space. This option controls how often this happens. If unspecified, *raved* will purge the visualization cache every two hours.
- threads** The number of worker threads RAVE spawns. Administrators should pick a number of threads based on available system resources and performance requirements. The default is five threads.
- namespace-config** Location of the analysis namespace configuration file. See Section 15.1 for more information.
- namespace-root** The optional root of all analysis namespace paths. See Section 15.1 for more information.
- log-config** Location of the logging configuration file.
- load-error-fatal** This option takes no value. If specified, errors encountered while loading analysis namespaces will cause *raved* to halt. Default behavior is to start anyway with all successfully-loaded analyses.
- url-base** When RAVE returns a ticket to a visualization, it prepends the contents of this option to the location of the visualization to create (usually) a URL.
- no-daemon** This option takes no value. If specified, *raved* will remain attached to the calling terminal and send its output to the console. This can be useful when troubleshooting. By default, *raved* detaches from the terminal and runs as a daemon process.

**--working-dir** When invoked as a daemon (see `--no-daemon`) *raved* will make the value of this option its working directory. If `--no-daemon` is supplied, the working directory will be the invoking user's current working directory and this value will be ignored. By default, *raved* will change its working directory to the root ("/").

**--pidfile** File to which *raved* should write its process ID.

### 14.2.2 Namespace configuration

For a fuller discussion of namespaces, see Section [15.1](#)

*raved* reads its namespace configuration from the namespace configuration file, whose name it gets from the `--namespace-config` command-line option. The configuration file has an INI-file syntax.

```
[namespaces]
foo=/usr/local/home/foo/namespace
bar=/usr/local/home/bar/namespace
```

This configuration maps the *foo* namespace to the directory `/usr/local/home/foo/namespace` and the *bar* namespace to `/usr/local/home/bar/namespace`. A request for *foo/op1* will look for the operation labeled *op1* in the *foo* namespace. A request for *baz/op1* or just */op1* would both yield errors.

Administrators may wish to use the *nsroot* variable to simplify the configuration file. The previous configuration file could be rewritten using *nsroot* in the following way:

```
[namespaces]
foo=%(nsroot)s/foo/namespace
bar=%(nsroot)s/bar/namespace
```

The string `%(nsroot)s` is replaced with the value of the `--namespace-root` command-line option.<sup>6</sup> If many or all of your namespaces share a common parent directory, this saves some typing, and may offer you some flexibility. (If the mount point for home directories changes, for instances, you don't need to modify the namespace configuration file; just restart *raved* with a different `--namespace-root`.)

### 14.2.3 Log configuration

*raved* reads its logging configuration from the file specified by the `--log-config` command-line option. *raved* uses the standard Python *logging* module to emit log information; for more information on the log configuration file syntax, see <http://www.python.org/doc/2.4.4/lib/module-logging.html>

## 14.3 *mod\_rave* configuration

*mod\_rave* is an Apache module; its configuration is done inside the Apache configuration file. For more information on Apache configuration file syntax, see <http://httpd.apache.org/docs/>.

To enable *mod\_rave*, you need to make sure *mod\_python* can locate the Python libraries needed to run it. To do this, use the *PythonPath* Apache configuration directive

---

<sup>6</sup>The `"%( )s"` syntax is an extension of the common `"printf"` syntax, and is common in Python. See <http://www.python.org/doc/2.4.4/lib/typesseq-strings.html> for more information on this pattern.

```
PythonPath "sys.path + [/path/to/the_module/]"
```

Next, add a location that *mod\_rave* will handle, as in this example:

```
<Location "/rave-proxy">
    SetHandler mod_python
    PythonHandler mod_rave
    PythonOption rave-service http://rave.example.com:8888/
</Location>
```

The *SetHandler* and *PythonHandler* directives tell Apache that *mod\_rave* will process requests made to this location. The two *PythonOption* directive specifies the location of the listening *raved* process.

## 15 Writing analyses for *raved*

To use RAVE operations with *raved*, they must supply typemaps and MIME types with the *@typemap* and *@mime\_type* decorators. The operations themselves may also require libraries to be installed on the system running *raved*.

Other than that, there are two other important concepts involved in running operations through *raved*—operation namespaces and exporting operations.

### 15.1 Namespaces

All operations accessible via *raved* live in *namespaces*. All *raved* requests have the form *<namespace>/<operation>*, where *<namespace>* is the namespace name, and *<operation>* is a symbolic name given to a RAVE operation.

Namespaces solve two problems that come up when running code in a server process such as *raved*—access control and availability.

Operations are basically code in Python modules. Normally, Python loads its modules once when its process first starts. It generally loads from only a few places, and access requirements for those locations are very simple—the user starting the process should be able to read from everywhere.

The assumptions may not hold in a long-running server process. The access requirements for *raved* are more like those of a web-server; several different people may contribute operations, and they need areas where they can read and write their code. However, they probably don't need access to areas where other people contribute their code.

Also like a web server, users contributing operations need to be able to change operations and see those changes without restarting the *raved* process. This allows administrators to separate the privilege of writing *raved*-accessible operations from the privileges of running or restarting the *raved* process.

Each *raved* namespace is a separate directory. *raved* treats this directory as a Python package (a Python module that can contain other modules). The maintainer of this namespace may organize the package however they wish. When a namespace maintainer makes a change to the namespace, *raved* recognizes the change and reloads the code.

Namespace maintainers don't need to be able to start or stop *raved* or change any of its configuration files, nor do they need to be able to see any of the other namespaces. The user running *raved* needs only read access to the namespaces.



## 15.2 Internal namespace organization

Internally, namespaces are Python *packages*—Python modules which contain other modules. Practically, this means the namespace directory contains an `__init__.py` file. This file can contain all the code for the namespace; more commonly, however, it is nearly empty, and contains only the definition of the `__all__` variable:

```
__all__ = ['amod', 'bmod']
```

This line tells Python that `amod.py` and `bmod.py` are part of this package, and should be loaded as well.

For more details on maintaining a Python package, see <http://www.python.org/doc/2.4.4/tut/node8.html>.

## 15.3 Exporting Operations

For an external client to request an analysis operation from *raved*, the operation must be *exported*. Other code running in *raved* may still call unexported operations and RAVE will cache their results normally, but they are not externally accessible.

To export an operation, define a Python dictionary named `__export__` in the module where you define your operations:

```
from rave.plugins.decorators import *

@op
@typemap(str)
@mime_type("text/plain")
def hello(name):
    return "Hello, %s!" % name

@op
@typemap(str)
@mime_type("text/plain")
def goodbye(name):
    return "Goodbye, %s!" % name

@op
@typemap(int)
@mime_type("text/plain")
def fib(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)

__export__ = {
    'fibonacci': fib,
    'greeting/hello': hello,
```

```
'greeting/goodbye': goodbye }
```

The keys to `__export__` are names for the analysis, while the values are the operations themselves. Assuming these operations were defined in the namespace `foo`, these operations would now be externally accessible as `foo/fibonacci`, `foo/greeting/hello` and `foo/greeting/goodbye`.

An export name must be unique within the namespace. RAVE-enabled services will combine the exported name with the namespace name to locate the exported operation. Beyond that, export names can be any string. However, we recommend you limit yourself to alphanumeric, hyphen and underscore characters for export names.

## Part V

# Appendices

## A List of expiration strategies

The following expiration strategies are supplied with RAVE. For a fuller discussion of expiration strategies, see Section 10.

***forever*** Returns a constant value indicating that the result should always remain cached.

***nocache*** Returns a constant value indicating that the result should never be cached.

RAVE also provides some *strategy factories*. The results of these functions are expiration strategy functions specialized to certain situations.

***duration\_strategy(interval)*** Returns an expiration strategy which will return a value that is *interval* seconds after the cached item is generated. *interval* can be a number or a string containing one or more interval specifiers of the form `<number><unit>`, where `<number>` is an integer and `<unit>` is one of *d*, *h*, *m*, or *s* (for days, hours, minutes and seconds, respectively). For example, the following operation:

```
@op
@expires(duration_strategy('2h30m'))
def foo(a, b):
    # ...
```

will expire two and a half hours after it is created.

*crossover\_strategy(a, b, age, age\_arg)* returns strategy *a* if the time in *age\_arg* is more recent than *age*; otherwise it returns *b*. This is often useful for caching time-series data; it is often useful to refresh such data more frequently if it is very recent, as new information may about the time in question may still be coming in. For example, the following operation:

```
five_mins = duration_strategy('5m')
five_days = duration_strategy('5d')
@op
```

```
@expires(crossover_strategy(
    five_mins, five_days, 3*60*60, 'etime'))
def foo(a, b, stime, etime):
    # ...
```

will cache data with an end time newer than three hours for five minutes, and will cache data older than three hours for five days.

## B List of RAVE type specifiers

Type specifiers are used in typemaps. See Section 12 for a discussion of what typemaps are and why they are useful.

The following is a list of type specifiers supplied by RAVE.

***list\_of(\*types)*** Specifies a list containing the types in *\*types*, in that order. If there are more items in the input list than in *\*types*, the *\*types* list is recycled to consume the input.

***one\_of(\*types)*** Specifies that the input should be converted successfully into at least one of the types listed in *\*types*. The returned value will be the output of the first successful conversion. One useful way to use this is to specify that an input can be either a value of a given type or *None*. For instance, a value that can be either a string or *None* could be represented by *one\_of(None, str)*. (Note that, because *str* would “successfully” convert the value *None* into the string “None”, the literal *None* has to precede it in the list.

***any*** Always succeeds, and returns the input value, unchanged. Use if a typemap is desirable, but the type of a particular parameter is unimportant.