

RAVE Administrator's Guide

CERT Network Situational Awareness Group

October 6, 2006

Part I

Introduction

This document will get you started with RAVE, the Retrospective Analysis and Visualization Engine. RAVE provides a common platform on which to develop and deploy analyses and visualizations—processes which transform and display collected data. These operations can be quite expensive; RAVE strives to ease the pain of analyzing and visualizing data without diminishing any of their benefits.

We will walk through installation and configuration of `raved`, RAVE's network interface; describe RAVE's network protocol; and walk through the process of developing new RAVE analyses.

RAVE is not a single application, but a system of components. `raved` and `mod_rave` (an Apache-based proxy for `raved` connection) provide RAVE analyses and visualizations over the network. Applications can also use the RAVE service directly through its core libraries. The RAVE analysis libraries make it easier to write good analyses and visualizations with RAVE.

1 Analysis as a service

RAVE provides analysis and visualization as a service, independent of any particular database or presentation application. This decoupling has several benefits:

- Developers and analysts can develop new analyses and visualizations independently of UI designers
- Analysis and visualization can be expensive operations. Using RAVE, system administrators can do this work on an "analysis server" and users can view the results on much lighter-weight workstations.
- Analyses can be shared as code between different organizations using RAVE.

RAVE provides two service interfaces: HTTP over sockets and in a local process space using RAVE API and operations directly.

2 Caching

RAVE caches the results of both analyses (as data to be re-used internally) and visualizations (as external file types which presentation applications can display). RAVE works very hard to make caching transparent

to all users of the system, including RAVE analysis developers. (If you develop analyses, however, you can take advantage of a few tricks to make your analyses cache more efficiently.)

Part II

Getting Started

3 Requirements

The RAVE core requires only Python 2.4 or higher at this time. You may require additional libraries for individual RAVE analysis plugins.

The RAVE analysis libraries rely heavily on matplotlib, a Python library for generating scientific graphics. `mod_rave`, the Apache proxy for `raved`, requires Apache 2 with `mod_python` linked against Python 2.4 or higher.

4 Installing RAVE

4.1 How RAVE is distributed

RAVE is distributed in two flavors: a Python distutils-based package and as a raw source archive. (RAVE distributions include the version number in the name of the distribution file, noted in the directions as `<version>`.)

4.1.1 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 `python-<version>.tar.gz`.

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

```
tar xvzf python-<version>.tar.gz
cd python-<version>
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-<version>.tar.gz` archive. The contents of `rave-extras` is outlined in Section 4.2.

4.1.2 Raw archive

If it is unsuitable to install from either a Python distutils package, we also distribute RAVE as a plain gzipped TAR archive. To distinguish between this and the Python distutils package, this file is named `rave-unpackaged-<version>.tar.gz`.

To install from the raw source files, extract the archive into some temporary space. This will create a directory called `rave-<version>`.

For more information on the contents of this archive, see Section 4.2.

4.2 RAVE Components

4.2.1 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.

4.2.2 `raved.py`

`raved.py` provides RAVE services via HTTP. See Section 5 for more information.

In addition, an example UNIX init script for running `raved.py` as a service is in the `examples` directory

4.2.3 The RAVE Apache module

`mod_rave.py` interfaces `raved` with Apache in much the same way that `Tomcat` interfaces with Apache through Apache modules. We recommend that you use this module for secure access to `raved`. See Section 7.1 for details.

4.2.4 RAVE configuration files

The `doc/examples` directory contains example log and namespace configuration files. For more information on these files, see Section 6.3 and Section 6.2. Also included is a sample init script for managing `raved` as a system service.

4.2.5 RAVE sample binaries

RAVE includes an `bin/samples` directory for 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 retrieves analyses from `raved`.

4.2.6 Documentation

You will find documentation in the `doc` directory of the unpackaged archive, including this document. There are also example configuration files for `raved.py` in the `examples` directory.

5 Running raved

The `raved.py` script (raved, for short) provides the RAVE service to the network. We often refer to raved's results as *visualizations* for historical reasons—at one time, that's all it could return. However, RAVE can now produce and return anything that it can store in a file, both images and raw data.

A client makes a request from raved the same way it makes any HTTP request. raved does not, however, respond directly with the result. This is because the result may not yet exist and may take a long time to generate; blocking while RAVE generates the visualization may be appropriate for some applications, but not for others.

Instead, raved responds with an XML document giving, among other things, the path to the file in which the results are stored. By default, this file may not exist when this XML document is given to the client. (The client can ask raved to block until it has created the visualization.) It is up to the client to retrieve the actual visualization. (The `mod_rave` proxy hides this 2-phase interaction from end-users. See Section 7 for more on `mod_rave`.)

6 Configuring raved

You can configure raved using command-line switches. In addition, raved uses configuration files to track its namespaces and logging.

6.1 Configuration options

raved takes the following options on the command line.

--ana-threads Synonym for `--threads`

--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`

--listen-addr raved will listen for connections on the interface bound to this network address. Section 6 discusses the security implications of this option in detail.

--listen-port raved will listen for connections to this port.

--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.

--log-config Location of the logging configuration file.

--namespace-config Location of the analysis namespace configuration file. See Section 6.2 for more information.

--namespace-root The optional root of all analysis namespace paths. See Section 6.2 for more information.

--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.

--output-dir Directory where raved should write visualizations.

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

--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.

--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.

--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.

--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 ("/).

6.2 Configuring Namespaces

Namespaces are a way to manage different sets of analyses. Some raved installations will have only one namespace, but others may have several. You may want to break the analyses up by the analyst or group that maintains them, for example.

The namespace configuration file maps namespaces to paths on the filesystem. When a request comes in to raved, raved uses the first part of the path specification (up to the first slash) as the namespace, and looks up the corresponding path. If the namespace exists, raved uses the rest of the path specification as the name of an analysis, which it looks up in the namespace.

A namespace itself is a Python package. See Section 10 for more information on how a namespace is organized.

The namespace configuration file has a Windows INI file-like syntax. (It's actually a Python ConfigParser syntax—see <http://docs.python.org/lib/module-ConfigParser.html> for more information.) The lines of the namespaces section map a namespace name on the left-hand side to a physical path on the right, for example:

```
[namespaces]
bob=/usr/home/bob/analyses
```

If you specify a *--namespace-root* configuration option, you may use the variable *nsroot* to refer to it in your configuration file:

```
[namespaces]
bob=%(nsroot)s/bob
alice=%(nsroot)s/alice
```

In this example, if you invoked *raved* with an *--namespace-root* option of */usr/local/analyses*, *raved* would search for the *bob* namespace in */usr/local/analyses/bob*. If you changed the location of analyses on your system, you could start raved with a different *namespace-root* option.

You can specify additional variables for use in the same way at the top of the configuration file:

```
analysts=/usr/local/analyst/namespaces
devs=/usr/local/developers/namespaces

[namespaces]
frank=%(analysts)s/frank
ernest=%(developers)s/ernest
```

6.3 Configuring Logging

The RAVE libraries and `raved.py` use the Python logging facility to regulate their log output. A sample log configuration script is included with the unpackaged archive in `bin/log.conf`. For more information on the syntax of this file and other aspects of Python logging, see <http://docs.python.org/lib/module-logging.html>, particularly <http://docs.python.org/lib/logging-config-fileformat.html>.

7 Using mod_rave

If you wish, you can configure `raved` to accept connections from remote hosts on a network. However, `raved` does not implement any security features like authentication, encryption or access control. For this reason, we recommend that you proxy the connection and provide these services on the proxy.

To make this process easier, RAVE comes with `mod_rave`, an Apache module that proxies `raved` connections. `mod_rave` is written in Python, and requires Apache 2 or later with `mod_python`, built against Python 2.4 or higher.

7.1 Configuring mod_rave

`mod_rave`'s configuration is similar to that of most Apache modules. We will assume the reader is familiar with Apache administration. For more information on configuring and maintaining Apache, consult the Apache documentation at <http://httpd.apache.org/docs/2.2/>. `mod_rave` is a `mod_python` module; you can get more information on `mod_python` at <http://modpython.org>.

To enable `mod_rave`, you need to make sure `mod_python` can locate the Python libraries needed to run it. To do this, insert the following line at the top of your Apache configuration file:

```
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` directives modify the behavior of `mod_rave`:

rave-service The URL to a RAVE service. Although this is the preferred method of configuring which RAVE to connect to, the following options also work. You must provide either *rave-service* or both of *rave-host* and *rave-port*.

rave-host The actual location of the machine hosting RAVE. This can be *localhost*, a resolvable hostname or an IP address.

rave-port The port on which RAVE will be listening. This is set in *raved* via the *--listen-port* command-line switch; the default for this option (in *mod_rave*) 8888.

Part III

Writing RAVE Analyses

One of the main design goals of RAVE was to make writing analyses as easy as possible. We hope that writing analyses in RAVE is a natural process that gets out of your way.

To distinguish them from normal Python functions, we call RAVE-enabled functions *operations*. In most respects, functions and operations are identical; you can call operations and functions from within each other, for instance.

There are two differences between normal functions and RAVE operations. First, operations are decorated (using Python's decorator syntax) so that RAVE can maintain the operation's cached results and state. (For more information on Python decorator syntax, see <http://www.python.org/dev/peps/pep-0318/> and <http://docs.python.org/ref/function.html>

Also, operations adhere to a certain contract, depending on the type of operation. For instance, RAVE assumes simple operations (decorated with *@op*) return all their results on the command line with no side effects, while file operations (decorated with *@op_file*) write their output to a file whose name it takes as an argument and return nothing.

8 A few simple examples

By way of example, here's how to write a Python function that returns the string "Hello, world!":

```
def hello():
    return "Hello, world!"
```

Here is the same thing as a RAVE analysis:

```
from rave.plugins.decorators import op
@op
def hello():
    return "Hello, world!"
```

As you can see, the only difference is that the function *hello* is decorated with *@op*, indicating to RAVE that it is an operation.

If you have RAVE installed, you can put the preceding code in a file (say, `test.py`) and run the following in an interactive session of the Python interpreter (lines preceded by “`???`” or “`...`” indicate lines you should type; normal lines indicate output from Python):

```
>>> from test import *
>>> hello()
'Hello, world'
```

Beyond testing that you have RAVE installed properly, this function doesn't do much to demonstrate RAVE's capabilities. Add the following operation to `test.py`:

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

Now run the following from the command-line:

```
>>> from test import *
>>> fib(50)
12586269025L
>>> fib(50)
12586269025L
>>> fib(25)
75025
```

Subjectively, you will notice that the second call to `fib(50)` took much less time than the first, and that the call to `fib(25)` was about as fast as the second call. You can see this 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.) The reason this function returns so quickly is because the results to all calls to `fib` are being cached. This includes not just the call to `fib(50)` which we make explicitly, but all the calls from `fib(0)` to `fib(49)` which `fib` itself makes to calculate `fib(50)`.

Incidentally, to get a feel for how much caching impacts the speed of this calculation, change your `fib` function to a normal Python function (by commenting out `@op` at the beginning of the function) and time it at the command line.


```
>>> from timeit import Timer
>>> 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
```

9 Exporting Operations

RAVE's caching capabilities are somewhat useful for exploration on the command-line or in scripts. However, RAVE analyses are most useful when service providers like *raved* can provide analysis results to other users. To do that, you need to export your operation.

Why explicitly export operations? We could export all operations, but just because a function is an operation does not mean external users should be able to run it. Exported operations can call unexported operations, and their results are cached like any other operation, so unexported operations still have great value. RAVE services need a way to know which operations to expose to users.

To export an operation, define a Python dictionary named `__export__` in the module where you define your operations. Let's do this to our *hello* and *fib* analyses. Here is the resulting *test.py*

```
from rave.plugins.decorators import op

@op
def hello():
    return "Hello, world!"

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

__export__ = {'fibonacci': fib, 'hello': hello }
```

The keys to `__export__` are string names for the analysis, while the values are the operations themselves. As you can see, the exported name of the function can be any name. There is only one restriction: an exported operation's name must be unique within the namespace. (See Section 10.) RAVE-enabled services will combine the exported name with the namespace name to locate the exported operation.

10 Working With Namespaces

Namespaces permit multiple people to write operations independently of one another. A namespace is a Python package which contains RAVE operations. RAVE-enabled services like *raved* use namespaces to

manage collections of operations independently of each other. The code within a namespace cannot access code in other namespaces, and names defined in one namespace will not overwrite those in another.

More technically, a namespace is a Python package—a directory which Python views as a kind of module that may contain other modules. (See <http://docs.python.org/tut/node8.html> for a more detailed description of Python packages.)

Let's now create a directory that we can use as a RAVE namespace. Create the directory `ns-test` and copy `test.py` into it, renaming it `__init__.py`. Here's what that might look like from a UNIX command prompt:

```
mkdir ns-test
cp test.py ns-test/__init__.py
```

This is sufficient to make your operations accessible to a RAVE service. For instance, you can split your code into multiple files or directories under `ns-test`. For example, create a file in `ns-test`, one named `hello.py` which contains our *hello* function.

```
from rave.plugins.decorators import op
@op
def hello():
    return "Hello, world!"

__export__ = {'hello': hello }
```

Create another named `fibonacci.py` which contains the *fib* function.

```
from rave.plugins.decorators import op

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

__export__ = {'fibonacci': fib}
```

(Note that each one imports `@op` and defines its own `__export__`.)

For RAVE to know that these operations are in these files, we need to modify `__init__.py`. Change `__init__.py` to the following:

```
__all__ = ['hello', 'fib']
```

"hello" and "fibonacci" are now module names, corresponding to the files we created. `__all__` defines the "public interface" of a Python module. Our public interface consists of these two modules.

`__all__` does not need to exist for Python to be able to use a module. RAVE, however, *does* need it to know that a module exists and contains exported operations.

You can define additional operations in `__init__.py` as well as specifying additional modules in this directory. You must define `__export__` in `__init__.py` which exports your operation. You don't have to change `__all__`, but you may want to add your operation to it, as that's in keeping with the intent of `__all__` in Python.

```
from rave.plugins.decorators import op

@op
def square(n):
    "Square a number."
    return n * n

__export__ = { 'square': square }

__all__ = ['hello', 'fib']
```

Note that this is a little confusing; we have some functions in the `__init__.py` of the package, and others in modules within the package. Generally, it's not a good idea to mix things this way. For maintainability's sake, keep things simple.

11 Operation options

You can specify the behavior of operations in a number of ways beyond defining the function that performs the operation. Some of these are very important to intelligently manage cached content and process arguments.

Passing options to an operation is simple. For instance, the `fib` operation should only ever receive numeric arguments. To enforce this, we can add a `typemap` option to `fib`'s `@op` declaration.

```
@op(typemap={'n': int})
def fib(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

mime_type RAVE-enabled applications use this to correctly report the type of data they are sending to clients. **Default:** "text/plain"

version A string that identifies an operation's version. RAVE uses `version` to identify when the code of an operation has changed, invalidating previously cached copies. When you change a function in a way that will change its results, change the version. **Default:** The modification time of the file in which the operation is defined. Note that adding a version to an operation that previously used the default is a version change, and will invalidate cached copies of the operation's results.

type_map A dictionary mapping function arguments by name to callables that coerce those arguments to a particular type. RAVE uses the type map to convert arguments from clients to the proper data type before executing an operation. **Default:** Unspecified arguments in a `typemap` are assumed to be of type `str` (string). An unspecified type map implies that all arguments are of type `str`.

cache_strat The value of this option determines how long results stay in the cache. RAVE comes with several cache strategies; the most commonly-used is the Duration strategy. For more information see Appendix A. **Default:** *never*: a cached item with this strategy never expires.

12 File Operations

@op is fine for analytic functions that can return their results directly to the caller. Sometimes, though, that is troublesome and inefficient.

Consider, for example, a set of analyses that use UNIX shell tools that use STDIN and STDOUT for input and output, which you chain together to perform complex analysis. To efficiently chain them together, you can use a subshell or the Python *subprocess* module.

If one of the intermediate steps of the analysis is time-consuming, however. You may want to make it a separate operation, so you can cache the data it returns and use it in subsequent analyses. Using *@op* requires running the command, reading its STDOUT into RAM and returning that. This might take up a lot of RAM while you read all STDOUT into memory. Alternately, you could have the command write to a file and return the name of the file from your analysis. But when the cache entry containing the filename expires, how do you delete the file?

RAVE's cache is file-based; if you could tell your shell process to write directly to the file RAVE would maintain in its cache, you could be maximally efficient and not have to worry about maintaining the file separately. This is exactly what *@op_file* provides.

Let's now write an analysis that uses *@op_file*. We'll make this one a little more practical. Suppose you want to analyze the netflow data of a single host (perhaps an important server) in many different ways using the SiLK tools. (SiLK is CERT NetSA's suite of tools for collecting and analyzing netflow. See for more information.) Let's write a file operation that selects the traffic related to this server from the main SiLK repository.

```
import os
import subprocess

@op_file
def filter_traffic(outfile, hostip, start, end):
    rwfiltercmd = """rwfilter --any-address=%s
        --start-date=%s --end-date=%s --pass=%s""" % (
        hostip, start, end, outfile)
    # I've omitted error-handling code for simplicity.
    retcode = call(rwfiltercmd)
```

Elsewhere in the code, you will want to use this file operation to pull the data of interest (or return it if it's already cached) and do additional processing on it. With an *@op* this is intuitive; it works a little bit differently with an *@op_file*. Here, we return the top ten hosts based on how many bytes the server sends to them over port 80. If this is a web server, this would give us some idea of who its heaviest individual users are. We will use the tools in *rave.plugins.shell* to simplify the process.

```
from rave.plugins.shell import pipe, parse_rwuniq

@op
def www_topten_clients(hostip, start, end):
```

```

filename = filter_traffic.run(hostip, start, end)

return parse_rwttotal(
    pipe(
        "rwfilter --sport 80"
        , "rwttotal --dip-first-24 --bytes"
        , "head -11" # 10 entries, plus header line
    )
)

```

The call to *filter_traffic.run* is remarkable because we defined *filter_traffic* as a function. However, the *@op_file* decorator replaced the function with a class instance—in this case, a *FileOperation*. *@op* does the same thing, replacing the function with an *Operation*. However, *@op*'s special caching behavior comes “for free” by calling the operation as if it were a function. With a file operation, we have to invoke a special method; calling a file operation like a function has the same effect as calling the function without any decoration.

The arguments to *run* are identical to those of the function, except without the first argument. For that, RAVE creates a filename and passes that in to the underlying function, expecting the function to write its output into that file. RAVE then manages that file in its cache; when the results of that operation should expire, it deletes the file.

A List of Cache Strategies

Cache strategies are classes that determine when a cached value should expire from RAVE's cache. Analysis developers can use one of the strategies that ship with RAVE, or they can build their own.

The following list describes the cache strategies that come with RAVE and their use.

In addition to the following classes, RAVE supplies two global cache strategy *instances* — *always* and *never*. An item with a strategy of *always* is always expired (in other words, it is a signal to RAVE not to cache this item). An item with a strategy of *never* never expires. *never* is the default strategy for operations and file operations; *always* is most useful in conjunction with composable strategies like *CrossoverStrategy*.

A.1 *Duration*

```
Duration(expire_interval)
```

Cached data will be valid until *expire_interval*, a duration in seconds, has elapsed.

A.2 *ActivityDuration*

```
ActivityDuration(activity_timeout, absolute_timeout=None)
```

Cached data will be valid until *activity_timeout*, a duration in seconds, has elapsed without someone requesting the item from the cache, or until *absolute_timeout* has elapsed, regardless of activity. If no absolute timeout is supplied, item can only expire from inactivity.

A.3 *CrossoverStrategy*

```
CrossoverStrategy(a_strat, b_strat, age, age_arg='etime')
```

The actual strategy for caching the data will be *a_strat* if the datetime value in the function parameter named by *age_arg* is younger than *age*. Otherwise, the strategy will be *b_strat*.

For example, consider the following:

```
dur_short = Duration(10)
dur_long = Duration(1000)

@op(cache_strat=CrossoverStrategy(dur_short, dur_long, 10, 'c'))
def foo(a, b, c):
    # ....

# Strategy will be dur_short
x = foo(1, 2, datetime.utcnow())

# Strategy will be dur_long
y = foo(1, 2, datetime.datetime(1900, 1, 1))
```

The value in *x* uses the *dur_short* cache strategy because the time in *c* is more recent than 10 seconds ago. The value in *y* will use the *dur_long* strategy; January 1, 1900 is considerably older than ten seconds ago.