supervisor Documentation

Release 4.0.0.dev0

Supervisor Developers

May 25, 2017

Contents

Supervisor is a client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems.

It shares some of the same goals of programs like *launchd*, *daemontools*, and *runit*. Unlike some of these programs, it is not meant to be run as a substitute for `init` as "process id 1". Instead it is meant to be used to control processes related to a project or a customer, and is meant to start like any other program at boot time.

CHAPTER 1

Narrative Documentation

Introduction

Overview

Supervisor is a client/server system that allows its users to control a number of processes on UNIX-like operating systems. It was inspired by the following:

Convenience

> It is often inconvenient to need to write `rc.d` scripts for every single process instance. `rc.d` scripts are a great lowest-common-denominator form of process initialization/autostart/management, but they can be painful to write and maintain. Additionally, `rc.d` scripts cannot automatically restart a crashed process and many programs do not restart themselves properly on a crash. Supervisord starts processes as its subprocesses, and can be configured to automatically restart them on a crash. It can also automatically be configured to start processes on its own invocation.

Accuracy

> It's often difficult to get accurate up/down status on processes on UNIX. Pidfiles often lie. Supervisord starts processes as subprocesses, so it always knows the true up/down status of its children and can be queried conveniently for this data.

Delegation

> Users who need to control process state often need only to do that. They don't want or need full-blown shell access to the machine on which the processes are running. Processes which listen on "low" TCP ports often need to be started and restarted as the root user (a UNIX misfeature). It's usually the case that it's perfectly fine to allow "normal" people to stop or restart such a process, but providing them with shell access is often impractical, and providing them with root access or sudo access is often impossible. It's also (rightly) difficult to explain to them why this problem exists. If supervisord is started as root, it is possible to allow "normal" users to control such processes without needing to explain the intricacies of the problem to them. Supervisorctl allows a very limited form of access to the machine, essentially allowing users to see process status and control supervisord-controlled subprocesses by emitting "stop", "start", and "restart" commands from a simple shell or web UI.

Process Groups

Processes often need to be started and stopped in groups, sometimes even in a "priority order". It's often difficult to explain to people how to do this. Supervisor allows you to assign priorities to processes, and allows user to emit commands via the supervisorctl client like "start all", and "restart all", which starts them in the preassigned priority order. Additionally, processes can be grouped into "process groups" and a set of logically related processes can be stopped and started as a unit.

Features

Simple

Supervisor is configured through a simple INI-style config file that's easy to learn. It provides many per-process options that make your life easier like restarting failed processes and automatic log rotation.

Centralized

Supervisor provides you with one place to start, stop, and monitor your processes. Processes can be controlled individually or in groups. You can configure Supervisor to provide a local or remote command line and web interface.

Efficient

Supervisor starts its subprocesses via fork/exec and subprocesses don't daemonize. The operating system signals Supervisor immediately when a process terminates, unlike some solutions that rely on troublesome PID files and periodic polling to restart failed processes.

Extensible

Supervisor has a simple event notification protocol that programs written in any language can use to monitor it, and an XML-RPC interface for control. It is also built with extension points that can be leveraged by Python developers.

Compatible

Supervisor works on just about everything except for Windows. It is tested and supported on Linux, Mac OS X, Solaris, and FreeBSD. It is written entirely in Python, so installation does not require a C compiler.

Proven

While Supervisor is very actively developed today, it is not new software. Supervisor has been around for years and is already in use on many servers.

Supervisor Components

`supervisord`

The server piece of supervisor is named **`supervisord`**. It is responsible for starting child programs at its own invocation, responding to commands from clients, restarting crashed or exited subprocesseses, logging its subprocess `stdout` and `stderr` output, and generating and handling "events" corresponding to points in subprocess lifetimes.

The server process uses a configuration file. This is typically located in `/etc/supervisord.conf`. This configuration file is a "Windows-INI" style config file. It is important to keep this file secure via proper filesystem permissions because it may contain unencrypted usernames and passwords.

`supervisorctl`

The command-line client piece of the supervisor is named **supervisorctl**. It provides a shell-like interface to the features provided by **supervisord**. From **supervisorctl**, a user can connect to different **supervisord** processes, get status on the subprocesses controlled by, stop and start subprocesses of, and get lists of running processes of a **supervisord**.

The command-line client talks to the server across a UNIX domain socket or an internet (TCP) socket. The server can assert that the user of a client should present authentication credentials before it allows him to perform commands. The client process typically uses the same configuration file as the server but any configuration file with a `[supervisorctl]` section in it will work.

Web Server

A (sparse) web user interface with functionality comparable to **supervisorctl** may be accessed via a browser if you start **supervisord** against an internet socket. Visit the server URL (e.g. `http://localhost:9001/`) to view and control process status through the web interface after activating the configuration file's `[inet_http_server]` section.

XML-RPC Interface

The same HTTP server which serves the web UI serves up an XML-RPC interface that can be used to interrogate and control supervisor and the programs it runs. See *XML-RPC API Documentation*.

Platform Requirements

Supervisor has been tested and is known to run on Linux (Ubuntu 9.10), Mac OS X (10.4/10.5/10.6), and Solaris (10 for Intel) and FreeBSD 6.1. It will likely work fine on most UNIX systems.

Supervisor will *not* run at all under any version of Windows.

Supervisor works under Python 2 version 2.6 or greater and Python 3 version 3.2 or greater.

Installing

Installation instructions depend whether the system on which you're attempting to install Supervisor has internet access.

Installing to A System With Internet Access

If your system has internet access, you can get Supervisor installed in two ways:

- Using `easy_install`, which is a feature of setuptools. This is the preferred method of installation.
- By downloading the Supervisor package and invoking a command.

Internet-Installing With Setuptools

If the Python interpreter you're using has Setuptools installed, and the system has internet access, you can download and install supervisor in one step using `easy_install`.

```
easy_install supervisor
```

Depending on the permissions of your system's Python, you might need to be the root user to install Supervisor successfully using `easy_install`.

Internet-Installing Without Setuptools

If your system does not have setuptools installed, you will need to download the Supervisor distribution and install it by hand. Current and previous Supervisor releases may be downloaded from PyPi. After unpacking the software archive, run `python setup.py install`. This requires internet access. It will download and install all distributions depended upon by Supervisor and finally install Supervisor itself.

Note: Depending on the permissions of your system's Python, you might need to be the root user to successfully invoke `python setup.py install`.

Installing To A System Without Internet Access

If the system that you want to install Supervisor to does not have Internet access, you'll need to perform installation slightly differently. Since both `easy_install` and `python setup.py install` depend on internet access to perform downloads of dependent software, neither will work on machines without internet access until dependencies are installed. To install to a machine which is not internet-connected, obtain the following dependencies on a machine which is internet-connected:

- setuptools (latest) from http://pypi.python.org/pypi/setuptools.
- meld3 (latest) from http://www.plope.com/software/meld3/.

Copy these files to removable media and put them on the target machine. Install each onto the target machine as per its instructions. This typically just means unpacking each file and invoking `python setup.py install` in the unpacked directory. Finally, run supervisor's `python setup.py install`.

Note: Depending on the permissions of your system's Python, you might need to be the root user to invoke `python setup.py install` successfully for each package.

Installing a Distribution Package

Some Linux distributions offer a version of Supervisor that is installable through the system package manager. These packages are made by third parties, not the Supervisor developers, and often include distribution-specific changes to Supervisor.

Use the package management tools of your distribution to check availability; e.g. on Ubuntu you can run `apt-cache show supervisor`, and on CentOS you can run `yum info supervisor`.

A feature of distribution packages of Supervisor is that they will usually include integration into the service management infrastructure of the distribution, e.g. allowing `supervisord` to automatically start when the system boots.

Note: Distribution packages of Supervisor can lag considerably behind the official Supervisor packages released to PyPI. For example, Ubuntu 12.04 (released April 2012) offered a package based on Supervisor 3.0a8 (released January 2010).

Note: Users reported that the distribution package of Supervisor for Ubuntu 16.04 had different behavior than previous versions. On Ubuntu 10.04, 12.04, and 14.04, installing the package will configure the system to start `supervisord` when the system boots. On Ubuntu 16.04, this was not done by the initial release of the package. The package was fixed later. See Ubuntu Bug #1594740 for more information.

Installing via pip

Supervisor can be installed with `pip install`:

```
pip install supervisor
```

Creating a Configuration File

Once the Supervisor installation has completed, run `echo_supervisord_conf`. This will print a "sample" Supervisor configuration file to your terminal's stdout.

Once you see the file echoed to your terminal, reinvoke the command as `echo_supervisord_conf > /etc/supervisord.conf`. This won't work if you do not have root access.

If you don't have root access, or you'd rather not put the `supervisord.conf` file in `/etc/supervisord.conf`, you can place it in the current directory (`echo_supervisord_conf > supervisord.conf`) and start **`supervisord`** with the `-c` flag in order to specify the configuration file location.

For example, `supervisord -c supervisord.conf`. Using the `-c` flag actually is redundant in this case, because **`supervisord`** searches the current directory for a `supervisord.conf` before it searches any other locations for the file, but it will work. See *Running Supervisor* for more information about the `-c` flag.

Once you have a configuration file on your filesystem, you can begin modifying it to your liking.

Running Supervisor

This section makes reference to a `BINDIR` when explaining how to run the **`supervisord`** and **`supervisorctl`** commands. This is the "bindir" directory that your Python installation has been configured with. For example, for an installation of Python installed via `./configure --prefix=/usr/local/py; make; make install`, `BINDIR` would be `/usr/local/py/bin`. Python interpreters on different platforms use a different `BINDIR`. Look at the output of `setup.py install` if you can't figure out where yours is.

Adding a Program

Before **`supervisord`** will do anything useful for you, you'll need to add at least one `program` section to its configuration. The `program` section will define a program that is run and managed when you invoke the **`supervisord`** command. To add a program, you'll need to edit the `supervisord.conf` file.

One of the simplest possible programs to run is the UNIX **`cat`** program. A `program` section that will run `cat` when the **`supervisord`** process starts up is shown below.

```
[program:foo]
command=/bin/cat
```

This stanza may be cut and pasted into the `supervisord.conf` file. This is the simplest possible program configuration, because it only names a command. Program configuration sections have many other configuration options which aren't shown here. See *[program:x] Section Settings* for more information.

Running `supervisord`

To start **`supervisord`**, run `$BINDIR/supervisord`. The resulting process will daemonize itself and detach from the terminal. It keeps an operations log at `$CWD/supervisor.log` by default.

You may start the **supervisord** executable in the foreground by passing the `-n` flag on its command line. This is useful to debug startup problems.

Warning: When **supervisord** starts up, it will search for its configuration file in default locations *including the current working directory*. If you are security-conscious you will probably want to specify a "-c" argument after the **supervisord** command specifying an absolute path to a configuration file to ensure that someone doesn't trick you into running supervisor from within a directory that contains a rogue `supervisord.conf` file. A warning is emitted when supervisor is started as root without this `-c` argument.

To change the set of programs controlled by **supervisord**, edit the `supervisord.conf` file and `kill -HUP` or otherwise restart the **supervisord** process. This file has several example program definitions.

The **supervisord** command accepts a number of command-line options. Each of these command line options overrides any equivalent value in the configuration file.

`supervisord` Command-Line Options

-c FILE, --configuration=FILE The path to a **supervisord** configuration file.

-n, --nodaemon Run **supervisord** in the foreground.

-h, --help Show **supervisord** command help.

-u USER, --user=USER UNIX username or numeric user id. If **supervisord** is started as the root user, setuid to this user as soon as possible during startup.

-m OCTAL, --umask=OCTAL Octal number (e.g. 022) representing the *umask* that should be used by **supervisord** after it starts.

-d PATH, --directory=PATH When supervisord is run as a daemon, cd to this directory before daemonizing.

-l FILE, --logfile=FILE Filename path to use as the supervisord activity log.

-y BYTES, --logfile_maxbytes=BYTES Max size of the supervisord activity log file before a rotation occurs. The value is suffix-multiplied, e.g "1" is one byte, "1MB" is 1 megabyte, "1GB" is 1 gigabyte.

-y NUM, --logfile_backups=NUM Number of backup copies of the supervisord activity log to keep around. Each logfile will be of size `logfile_maxbytes`.

-e LEVEL, --loglevel=LEVEL The logging level at which supervisor should write to the activity log. Valid levels are `trace`, `debug`, `info`, `warn`, `error`, and `critical`.

-j FILE, --pidfile=FILE The filename to which supervisord should write its pid file.

-i STRING, --identifier=STRING Arbitrary string identifier exposed by various client UIs for this instance of supervisor.

-q PATH, --childlogdir=PATH A path to a directory (it must already exist) where supervisor will write its `AUTO` -mode child process logs.

-k, --nocleanup Prevent **supervisord** from performing cleanup (removal of old `AUTO` process log files) at startup.

-a NUM, --minfds=NUM The minimum number of file descriptors that must be available to the supervisord process before it will start successfully.

-t, --strip_ansi Strip ANSI escape sequences from all child log process.

-v, --version Print the supervisord version number out to stdout and exit.

--profile_options=LIST Comma-separated options list for profiling. Causes **supervisord** to run under a profiler, and output results based on the options, which is a comma-separated list of the following: `cumulative`, `calls`, `callers`. E.g. `cumulative,callers`.

--minprocs=NUM The minimum number of OS process slots that must be available to the supervisord process before it will start successfully.

Running `supervisorctl`

To start **supervisorctl**, run `$BINDIR/supervisorctl`. A shell will be presented that will allow you to control the processes that are currently managed by **supervisord**. Type "help" at the prompt to get information about the supported commands.

The **supervisorctl** executable may be invoked with "one time" commands when invoked with arguments from a command line. An example: `supervisorctl stop all`. If arguments are present on the command-line, it will prevent the interactive shell from being invoked. Instead, the command will be executed and `supervisorctl` will exit with a code of 0 for success or running and non-zero for error. An example: `supervisorctl status all` would return non-zero if any single process was not running.

If **supervisorctl** is invoked in interactive mode against a **supervisord** that requires authentication, you will be asked for authentication credentials.

`supervisorctl` Command-Line Options

-c, --configuration Configuration file path (default /etc/supervisord.conf)

-h, --help Print usage message and exit

-i, --interactive Start an interactive shell after executing commands

-s, --serverurl URL URL on which supervisord server is listening (default "http://localhost:9001").

-u, --username Username to use for authentication with server

-p, --password Password to use for authentication with server

-r, --history-file Keep a readline history (if readline is available)

action [arguments]

Actions are commands like "tail" or "stop". If -i is specified or no action is specified on the command line, a "shell" interpreting actions typed interactively is started. Use the action "help" to find out about available actions.

`supervisorctl` Actions

help

Print a list of available actions

help <action>

Print help for <action>

add <name> [...]

Activates any updates in config for process/group

remove <name> [...]

Removes process/group from active config

update

Reload config and then add and remove as necessary (restarts programs)

clear <name>

Clear a process' log files.

clear <name> <name>

Clear multiple process' log files

clear all

Clear all process' log files

fg <process>

Connect to a process in foreground mode Press Ctrl+C to exit foreground

pid

Get the PID of supervisord.

pid <name>

Get the PID of a single child process by name.

pid all

Get the PID of every child process, one per line.

reread

Reload the daemon's configuration files, without add/remove (no restarts)

restart <name>

Restart a process Note: restart does not reread config files. For that, see reread and update.

restart <gname>:*

Restart all processes in a group Note: restart does not reread config files. For that, see reread and update.

restart <name> <name>

Restart multiple processes or groups Note: restart does not reread config files. For that, see reread and update.

restart all

Restart all processes Note: restart does not reread config files. For that, see reread and update.

signal

No help on signal

start <name>

Start a process

start <gname>:*

Start all processes in a group

start <name> <name>

Start multiple processes or groups

start all

Start all processes

status

Get all process status info.

status <name>

Get status on a single process by name.

status <name> <name>

Get status on multiple named processes.

stop <name>

Stop a process

stop <gname>:*

Stop all processes in a group

stop <name> <name>

Stop multiple processes or groups

stop all

Stop all processes

tail [-f] <name> [stdout|stderr] (default stdout)

Output the last part of process logs Ex: tail -f <name> Continuous tail of named process stdout Ctrl-C to exit. tail -100 <name> last 100 *bytes* of process stdout tail <name> stderr last 1600 *bytes* of process stderr

Signals

The **supervisord** program may be sent signals which cause it to perform certain actions while it's running.

You can send any of these signals to the single **supervisord** process id. This process id can be found in the file represented by the `pidfile` parameter in the `[supervisord]` section of the configuration file (by default it's `$CWD/supervisord.pid`).

Signal Handlers

`SIGTERM`

supervisord and all its subprocesses will shut down. This may take several seconds.

`SIGINT`

supervisord and all its subprocesses will shut down. This may take several seconds.

`SIGQUIT`

supervisord and all its subprocesses will shut down. This may take several seconds.

`SIGHUP`

supervisord will stop all processes, reload the configuration from the first config file it finds, and start all processes.

`SIGUSR2`

supervisord will close and reopen the main activity log and all child log files.

Runtime Security

The developers have done their best to assure that use of a **supervisord** process running as root cannot lead to unintended privilege escalation. But **caveat emptor**. Supervisor is not as paranoid as something like DJ Bernstein's *daemontools*, inasmuch as **supervisord** allows for arbitrary path specifications in its configuration file to which data may be written. Allowing arbitrary path selections can create vulnerabilities from symlink attacks. Be careful when specifying paths in your configuration. Ensure that the **supervisord** configuration file cannot be read from or written to by unprivileged users and that all files installed by the supervisor package have "sane" file permission protection settings. Additionally, ensure that your `PYTHONPATH` is sane and that all Python standard library files have adequate file permission protections.

Running `supervisord` automatically on startup

If you are using a distribution-packaged version of Supervisor, it should already be integrated into the service management infrastructure of your distribution.

There are user-contributed scripts for various operating systems at: https://github.com/Supervisor/initscripts

There are some answers at Serverfault in case you get stuck: How to automatically start supervisord on Linux (Ubuntu)

Configuration File

The Supervisor configuration file is conventionally named `supervisord.conf`. It is used by both **supervisord** and **supervisorctl**. If either application is started without the `-c` option (the option which is used to tell the application the configuration filename explicitly), the application will look for a file named `supervisord.conf` within the following locations, in the specified order. It will use the first file it finds.

1. `$CWD/supervisord.conf`
2. `$CWD/etc/supervisord.conf`
3. `/etc/supervisord.conf`
4. `/etc/supervisor/supervisord.conf` (since Supervisor 3.3.0)
5. `../etc/supervisord.conf` (Relative to the executable)
6. `../supervisord.conf` (Relative to the executable)

Note: Many versions of Supervisor packaged for Debian and Ubuntu included a patch that added `/etc/supervisor/supervisord.conf` to the search paths. The first PyPI package of Supervisor to include it was Supervisor 3.3.0.

File Format

`supervisord.conf` is a Windows-INI-style (Python ConfigParser) file. It has sections (each denoted by a `[header]`) and key / value pairs within the sections. The sections and their allowable values are described below.

Environment Variables

Environment variables that are present in the environment at the time that **supervisord** is started can be used in the configuration file using the Python string expression syntax `%(ENV_X)s`:

```
[program:example]
command=/usr/bin/example --loglevel=%(ENV_LOGLEVEL)s
```

In the example above, the expression `%(ENV_LOGLEVEL)s` would be expanded to the value of the environment variable `LOGLEVEL`.

Note: In Supervisor 3.2 and later, `%(ENV_X)s` expressions are supported in all options. In prior versions, some options support them, but most do not. See the documentation for each option below.

`[unix_http_server]` Section Settings

The `supervisord.conf` file contains a section named `[unix_http_server]` under which configuration parameters for an HTTP server that listens on a UNIX domain socket should be inserted. If the configuration file has no `[unix_http_server]` section, a UNIX domain socket HTTP server will not be started. The allowable configuration values are as follows.

`[unix_http_server]` Section Values

`file`

A path to a UNIX domain socket (e.g. `/tmp/supervisord.sock`) on which supervisor will listen for HTTP/XML-RPC requests. **supervisorctl** uses XML-RPC to communicate with **supervisord** over this port. This option can include the value `%(here)s`, which expands to the directory in which the **supervisord** configuration file was found.

Default: None.

Required: No.

Introduced: 3.0

`chmod`

Change the UNIX permission mode bits of the UNIX domain socket to this value at startup.

Default: `0700`

Required: No.

Introduced: 3.0

`chown`

Change the user and group of the socket file to this value. May be a UNIX username (e.g. `chrism`) or a UNIX username and group separated by a colon (e.g. `chrism:wheel`).

Default: Use the username and group of the user who starts supervisord.

Required: No.

Introduced: 3.0

`username`

The username required for authentication to this HTTP server.

Default: No username required.

Required: No.

Introduced: 3.0

`password`

The password required for authentication to this HTTP server. This can be a cleartext password, or can be specified as a SHA-1 hash if prefixed by the string `{SHA}`. For example, `{SHA}82ab876d1387bfafe46cc1c8a2ef074eae50cb1d` is the SHA-stored version of the password "thepassword".

Note that hashed password must be in hex format.

Default: No password required.

Required: No.

Introduced: 3.0

`[unix_http_server]` Section Example

```
[unix_http_server]
file = /tmp/supervisor.sock
chmod = 0777
chown= nobody:nogroup
username = user
password = 123
```

`[inet_http_server]` Section Settings

The `supervisord.conf` file contains a section named `[inet_http_server]` under which configuration parameters for an HTTP server that listens on a TCP (internet) socket should be inserted. If the configuration file has no `[inet_http_server]` section, an inet HTTP server will not be started. The allowable configuration values are as follows.

`[inet_http_server]` Section Values

`port`

A TCP host:port value or (e.g. `127.0.0.1:9001`) on which supervisor will listen for HTTP/XML-RPC requests. **`supervisorctl`** will use XML-RPC to communicate with **`supervisord`** over this port. To listen on all interfaces in the machine, use `:9001` or `*:9001`.

Default: No default.

Required: Yes.

Introduced: 3.0

`username`

The username required for authentication to this HTTP server.

Default: No username required.

Required: No.

Introduced: 3.0

`password`

The password required for authentication to this HTTP server. This can be a cleartext password, or can be specified as a SHA-1 hash if prefixed by the string `{SHA}`. For example, `{SHA}82ab876d1387bfafe46cc1c8a2ef074eae50cb1d` is the SHA-stored version of the password "thepassword".

Note that hashed password must be in hex format.

Default: No password required.

Required: No.

Introduced: 3.0

`[inet_http_server]` Section Example

```
[inet_http_server]
port = 127.0.0.1:9001
username = user
password = 123
```

`[supervisord]` Section Settings

The `supervisord.conf` file contains a section named `[supervisord]` in which global settings related to the **`supervisord`** process should be inserted. These are as follows.

`[supervisord]` Section Values

`logfile`

The path to the activity log of the supervisord process. This option can include the value `%(here)s`, which expands to the directory in which the supervisord configuration file was found.

Default: `$CWD/supervisord.log`

Required: No.

Introduced: 3.0

`logfile_maxbytes`

The maximum number of bytes that may be consumed by the activity log file before it is rotated (suffix multipliers like "KB", "MB", and "GB" can be used in the value). Set this value to 0 to indicate an unlimited log size.

Default: 50MB

Required: No.

Introduced: 3.0

`logfile_backups`

The number of backups to keep around resulting from activity log file rotation. If set to 0, no backups will be kept.

Default: 10

Required: No.

Introduced: 3.0

`loglevel`

The logging level, dictating what is written to the supervisord activity log. One of `critical`, `error`, `warn`, `info`, `debug`, `trace`, or `blather`. Note that at log level `debug`, the supervisord log file will record the stderr/stdout output of its child processes and extended info info about process state changes, which is useful for debugging a process which isn't starting properly. See also: *Activity Log Levels*.

Default: info

Required: No.

Introduced: 3.0

`pidfile`

The location in which supervisord keeps its pid file. This option can include the value `%(here)s`, which expands to the directory in which the supervisord configuration file was found.

Default: `$CWD/supervisord.pid`

Required: No.

Introduced: 3.0

`umask`

The *umask* of the supervisord process.

Default: `022`

Required: No.

Introduced: 3.0

`nodaemon`

If true, supervisord will start in the foreground instead of daemonizing.

Default: false

Required: No.

Introduced: 3.0

`minfds`

The minimum number of file descriptors that must be available before supervisord will start successfully. A call to setrlimit will be made to attempt to raise the soft and hard limits of the supervisord process to satisfy `minfds`. The hard limit may only be raised if supervisord is run as root. supervisord uses file descriptors liberally, and will enter a failure mode when one cannot be obtained from the OS, so it's useful to be able to specify a minimum value to ensure it doesn't run out of them during execution. These limits will be inherited by the managed subprocesses. This option is particularly useful on Solaris, which has a low per-process fd limit by default.

Default: 1024

Required: No.

Introduced: 3.0

`minprocs`

The minimum number of process descriptors that must be available before supervisord will start successfully. A call to setrlimit will be made to attempt to raise the soft and hard limits of the supervisord process to satisfy `minprocs`. The hard limit may only be raised if supervisord is run as root. supervisord will enter a failure mode when the OS runs out of process descriptors, so it's useful to ensure that enough process descriptors are available upon **supervisord** startup.

Default: 200

Required: No.

Introduced: 3.0

`nocleanup`

Prevent supervisord from clearing any existing `AUTO` child log files at startup time. Useful for debugging.

Default: false

Required: No.

Introduced: 3.0

`childlogdir`

The directory used for `AUTO` child log files. This option can include the value `%(here)s`, which expands to the directory in which the **supervisord** configuration file was found.

Default: value of Python's `tempfile.get_tempdir()`

Required: No.

Introduced: 3.0

`user`

Instruct **supervisord** to switch users to this UNIX user account before doing any meaningful processing. The user can only be switched if **supervisord** is started as the root user. If **supervisord** can't switch users, it will still continue but will write a log message at the `critical` level saying that it can't drop privileges.

Default: do not switch users

Required: No.

Introduced: 3.0

`directory`

When **supervisord** daemonizes, switch to this directory. This option can include the value `%(here)s`, which expands to the directory in which the **supervisord** configuration file was found.

Default: do not cd

Required: No.

Introduced: 3.0

`strip_ansi`

Strip all ANSI escape sequences from child log files.

Default: false

Required: No.

Introduced: 3.0

`environment`

A list of key/value pairs in the form `KEY="val",KEY2="val2"` that will be placed in the **supervisord** process' environment (and as a result in all of its child process' environments). This option can include the value `%(here)s`, which expands to the directory in which the supervisord configuration file was found. Values containing non-alphanumeric characters should be quoted (e.g. `KEY="val:123",KEY2="val,456"`). Otherwise, quoting the values is optional but recommended. To escape percent characters, simply use two. (e.g. `URI="/first%%20name"`) **Note** that subprocesses will inherit the environment variables of the shell used to start **supervisord** except for the ones overridden here and within the program's `environment` option. See *Subprocess Environment*.

Default: no values

Required: No.

Introduced: 3.0

`identifier`

The identifier string for this supervisor process, used by the RPC interface.

Default: supervisor

Required: No.

Introduced: 3.0

`[supervisord]` Section Example

```
[supervisord]
logfile = /tmp/supervisord.log
logfile_maxbytes = 50MB
logfile_backups=10
loglevel = info
pidfile = /tmp/supervisord.pid
nodaemon = false
minfds = 1024
minprocs = 200
umask = 022
user = chrism
identifier = supervisor
directory = /tmp
nocleanup = true
childlogdir = /tmp
strip_ansi = false
environment = KEY1="value1",KEY2="value2"
```

`[supervisorctl]` Section Settings

The configuration file may contain settings for the **supervisorctl** interactive shell program. These options are listed below.

`[supervisorctl]` Section Values

`serverurl`

The URL that should be used to access the supervisord server, e.g. `http://localhost:9001`. For UNIX domain sockets, use `unix:///absolute/path/to/file.sock`.

Default: `http://localhost:9001`

Required: No.

Introduced: 3.0

`username`

The username to pass to the supervisord server for use in authentication. This should be same as `username` from the supervisord server configuration for the port or UNIX domain socket you're attempting to access.

Default: No username

Required: No.

Introduced: 3.0

`password`

The password to pass to the supervisord server for use in authentication. This should be the cleartext version of `password` from the supervisord server configuration for the port or UNIX domain socket you're attempting to access. This value cannot be passed as a SHA hash. Unlike other passwords specified in this file, it must be provided in cleartext.

Default: No password

Required: No.

Introduced: 3.0

`prompt`

String used as supervisorctl prompt.

Default: `supervisor`

Required: No.

Introduced: 3.0

`history_file`

A path to use as the `readline` persistent history file. If you enable this feature by choosing a path, your supervisorctl commands will be kept in the file, and you can use readline (e.g. arrow-up) to invoke commands you performed in your last supervisorctl session.

Default: No file

Required: No.

Introduced: 3.0a5

`[supervisorctl]` Section Example

```
[supervisorctl]
serverurl = unix:///tmp/supervisor.sock
username = chris
password = 123
prompt = mysupervisor
```

[program:x] Section Settings

The configuration file must contain one or more `program` sections in order for supervisord to know which programs it should start and control. The header value is composite value. It is the word "program", followed directly by a colon, then the program name. A header value of `[program:foo]` describes a program with the name of "foo". The name is used within client applications that control the processes that are created as a result of this configuration. It is an error to create a `program` section that does not have a name. The name must not include a colon character or a bracket character. The value of the name is used as the value for the `%(program_name)s` string expression expansion within other values where specified.

Note: A `[program:x]` section actually represents a "homogeneous process group" to supervisor (as of 3.0). The members of the group are defined by the combination of the `numprocs` and `process_name` parameters in the configuration. By default, if numprocs and process_name are left unchanged from their defaults, the group represented by `[program:x]` will be named `x` and will have a single process named `x` in it. This provides a modicum of backwards compatibility with older supervisor releases, which did not treat program sections as homogeneous process group definitions.

But for instance, if you have a `[program:foo]` section with a `numprocs` of 3 and a `process_name` expression of `%(program_name)s_%(process_num)02d`, the "foo" group will contain three processes, named `foo_00`, `foo_01`, and `foo_02`. This makes it possible to start a number of very similar processes using a single `[program:x]` section. All logfile names, all environment strings, and the command of programs can also contain similar Python string expressions, to pass slightly different parameters to each process.

[program:x] Section Values

`command`

The command that will be run when this program is started. The command can be either absolute (e.g. `/path/to/programname`) or relative (e.g. `programname`). If it is relative, the supervisord's environment `$PATH` will be searched for the executable. Programs can accept arguments, e.g. `/path/to/program foo bar`. The command line can use double quotes to group arguments with spaces in them to pass to the program, e.g. `/path/to/program/name -p "foo bar"`. Note that the value of `command` may include Python string expressions, e.g. `/path/to/programname --port=80%(process_num)02d` might expand to `/path/to/programname --port=8000` at runtime. String expressions are evaluated against a dictionary containing the keys `group_name`, `host_node_name`, `process_num`, `program_name`, `here` (the directory of the supervisord config file), and all supervisord's environment variables prefixed with `ENV_`. Controlled programs should themselves not be daemons, as supervisord assumes it is responsible for daemonizing its subprocesses (see *Nondaemonizing of Subprocesses*).

Note: The command will be truncated if it looks like a config file comment, e.g. `command=bash -c 'foo ; bar'` will be truncated to `command=bash -c 'foo`. Quoting will not prevent this behavior, since the configuration file reader does not parse the command like a shell would.

Default: No default.

Required: Yes.

Introduced: 3.0

`process_name`

A Python string expression that is used to compose the supervisor process name for this process. You usually don't need to worry about setting this unless you change `numprocs`. The string expression

is evaluated against a dictionary that includes `group_name`, `host_node_name`, `process_num`, `program_name`, and `here` (the directory of the supervisord config file).

Default: `%(program_name)s`

Required: No.

Introduced: 3.0

`numprocs`

Supervisor will start as many instances of this program as named by numprocs. Note that if numprocs > 1, the `process_name` expression must include `%(process_num)s` (or any other valid Python string expression that includes `process_num`) within it.

Default: 1

Required: No.

Introduced: 3.0

`numprocs_start`

An integer offset that is used to compute the number at which `numprocs` starts.

Default: 0

Required: No.

Introduced: 3.0

`priority`

The relative priority of the program in the start and shutdown ordering. Lower priorities indicate programs that start first and shut down last at startup and when aggregate commands are used in various clients (e.g. "start all"/"stop all"). Higher priorities indicate programs that start last and shut down first.

Default: 999

Required: No.

Introduced: 3.0

`autostart`

If true, this program will start automatically when supervisord is started.

Default: true

Required: No.

Introduced: 3.0

`startsecs`

The total number of seconds which the program needs to stay running after a startup to consider the start successful (moving the process from the `STARTING` state to the `RUNNING` state). Set to `0` to indicate that the program needn't stay running for any particular amount of time.

Note: Even if a process exits with an "expected" exit code (see `exitcodes`), the start will still be considered a failure if the process exits quicker than `startsecs`.

Default: 1

Required: No.

Introduced: 3.0

`startretries`

The number of serial failure attempts that **supervisord** will allow when attempting to start the program before giving up and putting the process into an `FATAL` state. See *Process States* for explanation of the `FATAL` state.

Default: 3

Required: No.

Introduced: 3.0

`autorestart`

Specifies if **supervisord** should automatically restart a process if it exits when it is in the `RUNNING` state. May be one of `false`, `unexpected`, or `true`. If `false`, the process will not be autorestarted. If `unexpected`, the process will be restarted when the program exits with an exit code that is not one of the exit codes associated with this process' configuration (see `exitcodes`). If `true`, the process will be unconditionally restarted when it exits, without regard to its exit code.

Note: `autorestart` controls whether **supervisord** will autorestart a program if it exits after it has successfully started up (the process is in the `RUNNING` state).

supervisord has a different restart mechanism for when the process is starting up (the process is in the `STARTING` state). Retries during process startup are controlled by `startsecs` and `startretries`.

Default: unexpected

Required: No.

Introduced: 3.0

`exitcodes`

The list of "expected" exit codes for this program used with `autorestart`. If the `autorestart` parameter is set to `unexpected`, and the process exits in any other way than as a result of a supervisor stop request, **supervisord** will restart the process if it exits with an exit code that is not defined in this list.

Default: 0,2

Required: No.

Introduced: 3.0

`stopsignal`

The signal used to kill the program when a stop is requested. This can be any of TERM, HUP, INT, QUIT, KILL, USR1, or USR2.

Default: TERM

Required: No.

Introduced: 3.0

`stopwaitsecs`

The number of seconds to wait for the OS to return a SIGCHLD to **supervisord** after the program has been sent a stopsignal. If this number of seconds elapses before **supervisord** receives a SIGCHLD from the process, **supervisord** will attempt to kill it with a final SIGKILL.

Default: 10

Required: No.

Introduced: 3.0

`stopasgroup`

If true, the flag causes supervisor to send the stop signal to the whole process group and implies `killasgroup` is true. This is useful for programs, such as Flask in debug mode, that do not propagate stop signals to their children, leaving them orphaned.

Default: false

Required: No.

Introduced: 3.0b1

`killasgroup`

If true, when resorting to send SIGKILL to the program to terminate it send it to its whole process group instead, taking care of its children as well, useful e.g with Python programs using `multiprocessing`.

Default: false

Required: No.

Introduced: 3.0a11

`user`

Instruct **supervisord** to use this UNIX user account as the account which runs the program. The user can only be switched if **supervisord** is run as the root user. If **supervisord** can't switch to the specified user, the program will not be started.

Note: The user will be changed using `setuid` only. This does not start a login shell and does not change environment variables like `USER` or `HOME`. See *Subprocess Environment* for details.

Default: Do not switch users

Required: No.

Introduced: 3.0

`redirect_stderr`

If true, cause the process' stderr output to be sent back to **supervisord** on its stdout file descriptor (in UNIX shell terms, this is the equivalent of executing `/the/program 2>&1`).

Note: Do not set `redirect_stderr=true` in an `[eventlistener:x]` section. Eventlisteners use `stdout` and `stdin` to communicate with `supervisord`. If `stderr` is redirected, output from `stderr` will interfere with the eventlistener protocol.

Default: false

Required: No.

Introduced: 3.0, replaces 2.0's `log_stdout` and `log_stderr`

`stdout_logfile`

Put process stdout output in this file (and if redirect_stderr is true, also place stderr output in this file). If `stdout_logfile` is unset or set to `AUTO`, supervisor will automatically choose a file location. If this is set to `NONE`, supervisord will create no log file. `AUTO` log files and their backups will be deleted when **supervisord** restarts. The `stdout_logfile` value can contain Python string expressions that will evaluated against a dictionary that contains the keys `group_name`, `host_node_name`, `process_num`, `program_name`, and `here` (the directory of the supervisord config file).

Note: It is not possible for two processes to share a single log file (`stdout_logfile`) when rotation (`stdout_logfile_maxbytes`) is enabled. This will result in the file being corrupted.

Default: `AUTO`

Required: No.

Introduced: 3.0, replaces 2.0's `logfile`

`stdout_logfile_maxbytes`

The maximum number of bytes that may be consumed by `stdout_logfile` before it is rotated (suffix multipliers like "KB", "MB", and "GB" can be used in the value). Set this value to 0 to indicate an unlimited log size.

Default: 50MB

Required: No.

Introduced: 3.0, replaces 2.0's `logfile_maxbytes`

`stdout_logfile_backups`

The number of `stdout_logfile` backups to keep around resulting from process stdout log file rotation. If set to 0, no backups will be kept.

Default: 10

Required: No.

Introduced: 3.0, replaces 2.0's `logfile_backups`

`stdout_capture_maxbytes`

Max number of bytes written to capture FIFO when process is in "stdout capture mode" (see *Capture Mode*). Should be an integer (suffix multipliers like "KB", "MB" and "GB" can used in the value). If this value is 0, process capture mode will be off.

Default: 0

Required: No.

Introduced: 3.0

`stdout_events_enabled`

If true, PROCESS_LOG_STDOUT events will be emitted when the process writes to its stdout file descriptor. The events will only be emitted if the file descriptor is not in capture mode at the time the data is received (see *Capture Mode*).

Default: 0

Required: No.

Introduced: 3.0a7

`stdout_syslog`

If true, stdout will be directed to syslog along with the process name.

Default: False

Required: No.

Introduced: 4.0.0

`stderr_logfile`

Put process stderr output in this file unless `redirect_stderr` is true. Accepts the same value types as `stdout_logfile` and may contain the same Python string expressions.

Note: It is not possible for two processes to share a single log file (`stderr_logfile`) when rotation (`stderr_logfile_maxbytes`) is enabled. This will result in the file being corrupted.

Default: `AUTO`

Required: No.

Introduced: 3.0

`stderr_logfile_maxbytes`

The maximum number of bytes before logfile rotation for `stderr_logfile`. Accepts the same value types as `stdout_logfile_maxbytes`.

Default: 50MB

Required: No.

Introduced: 3.0

`stderr_logfile_backups`

The number of backups to keep around resulting from process stderr log file rotation. If set to 0, no backups will be kept.

Default: 10

Required: No.

Introduced: 3.0

`stderr_capture_maxbytes`

Max number of bytes written to capture FIFO when process is in "stderr capture mode" (see *Capture Mode*). Should be an integer (suffix multipliers like "KB", "MB" and "GB" can used in the value). If this value is 0, process capture mode will be off.

Default: 0

Required: No.

Introduced: 3.0

`stderr_events_enabled`

If true, PROCESS_LOG_STDERR events will be emitted when the process writes to its stderr file descriptor. The events will only be emitted if the file descriptor is not in capture mode at the time the data is received (see *Capture Mode*).

Default: false

Required: No.

Introduced: 3.0a7

`stderr_syslog`

If true, stderr will be directed to syslog along with the process name.

Default: False

Required: No.

Introduced: 4.0.0

`environment`

A list of key/value pairs in the form `KEY="val",KEY2="val2"` that will be placed in the child process' environment. The environment string may contain Python string expressions that will be evaluated against a dictionary containing `group_name`, `host_node_name`, `process_num`, `program_name`, and `here` (the directory of the supervisord config file). Values containing non-alphanumeric characters should be quoted (e.g. `KEY="val:123",KEY2="val,456"`). Otherwise, quoting the values is optional but recommended. **Note** that the subprocess will inherit the environment variables of the shell used to start "supervisord" except for the ones overridden here. See *Subprocess Environment*.

Default: No extra environment

Required: No.

Introduced: 3.0

`directory`

A file path representing a directory to which **`supervisord`** should temporarily chdir before exec'ing the child.

Default: No chdir (inherit supervisor's)

Required: No.

Introduced: 3.0

`umask`

An octal number (e.g. 002, 022) representing the umask of the process.

Default: No special umask (inherit supervisor's)

Required: No.

Introduced: 3.0

`serverurl`

The URL passed in the environment to the subprocess process as `SUPERVISOR_SERVER_URL` (see `supervisor.childutils`) to allow the subprocess to easily communicate with the internal HTTP server. If provided, it should have the same syntax and structure as the `[supervisorctl]` section option of the same name. If this is set to AUTO, or is unset, supervisor will automatically construct a server URL, giving preference to a server that listens on UNIX domain sockets over one that listens on an internet socket.

Default: AUTO

Required: No.

Introduced: 3.0

[program:x] Section Example

```
[program:cat]
command=/bin/cat
process_name=%(program_name)s
numprocs=1
directory=/tmp
umask=022
priority=999
autostart=true
autorestart=unexpected
startsecs=10
startretries=3
exitcodes=0,2
stopsignal=TERM
stopwaitsecs=10
stopasgroup=false
killasgroup=false
user=chrism
redirect_stderr=false
stdout_logfile=/a/path
stdout_logfile_maxbytes=1MB
stdout_logfile_backups=10
stdout_capture_maxbytes=1MB
stdout_events_enabled=false
stderr_logfile=/a/path
stderr_logfile_maxbytes=1MB
stderr_logfile_backups=10
stderr_capture_maxbytes=1MB
stderr_events_enabled=false
environment=A="1",B="2"
serverurl=AUTO
```

[include] Section Settings

The `supervisord.conf` file may contain a section named `[include]`. If the configuration file contains an `[include]` section, it must contain a single key named "files". The values in this key specify other configuration files to be included within the configuration.

[include] Section Values

`files`

A space-separated sequence of file globs. Each file glob may be absolute or relative. If the file glob is relative, it is considered relative to the location of the configuration file which includes it. A "glob" is a file pattern which matches a specified pattern according to the rules used by the Unix shell. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. The string expression is evaluated against a dictionary that includes `host_node_name` and `here` (the directory of the supervisord config file). Recursive includes from included files are not supported.

Default: No default (required)

Required: Yes.

Introduced: 3.0

Changed: 3.3.0. Added support for the `host_node_name` expansion.

[include] Section Example

```
[include]
files = /an/absolute/filename.conf /an/absolute/*.conf foo.conf config??.conf
```

[group:x] Section Settings

It is often useful to group "homogeneous" process groups (aka "programs") together into a "heterogeneous" process group so they can be controlled as a unit from Supervisor's various controller interfaces.

To place programs into a group so you can treat them as a unit, define a `[group:x]` section in your configuration file. The group header value is a composite. It is the word "group", followed directly by a colon, then the group name. A header value of `[group:foo]` describes a group with the name of "foo". The name is used within client applications that control the processes that are created as a result of this configuration. It is an error to create a `group` section that does not have a name. The name must not include a colon character or a bracket character.

For a `[group:x]`, there must be one or more `[program:x]` sections elsewhere in your configuration file, and the group must refer to them by name in the `programs` value.

If "homogeneous" process groups (represented by program sections) are placed into a "heterogeneous" group via `[group:x]` section's `programs` line, the homogeneous groups that are implied by the program section will not exist at runtime in supervisor. Instead, all processes belonging to each of the homogeneous groups will be placed into the heterogeneous group. For example, given the following group configuration:

```
[group:foo]
programs=bar,baz
priority=999
```

Given the above, at supervisord startup, the `bar` and `baz` homogeneous groups will not exist, and the processes that would have been under them will now be moved into the `foo` group.

[group:x] Section Values

`programs`

A comma-separated list of program names. The programs which are listed become members of the group.

Default: No default (required)

Required: Yes.

Introduced: 3.0

`priority`

A priority number analogous to a `[program:x]` priority value assigned to the group.

Default: 999

Required: No.

Introduced: 3.0

[group:x] Section Example

```
[group:foo]
programs=bar,baz
priority=999
```

[fcgi-program:x] Section Settings

Supervisor can manage groups of FastCGI processes that all listen on the same socket. Until now, deployment flexibility for FastCGI was limited. To get full process management, you could use mod_fastcgi under Apache but then you were stuck with Apache's inefficient concurrency model of one process or thread per connection. In addition to requiring more CPU and memory resources, the process/thread per connection model can be quickly saturated by a slow resource, preventing other resources from being served. In order to take advantage of newer event-driven web servers such as lighttpd or nginx which don't include a built-in process manager, you had to use scripts like cgi-fcgi or spawn-fcgi. These can be used in conjunction with a process manager such as supervisord or daemontools but require each FastCGI child process to bind to its own socket. The disadvantages of this are: unnecessarily complicated web server configuration, ungraceful restarts, and reduced fault tolerance. With fewer sockets to configure, web server configurations are much smaller if groups of FastCGI processes can share sockets. Shared sockets allow for graceful restarts because the socket remains bound by the parent process while any of the child processes are being restarted. Finally, shared sockets are more fault tolerant because if a given process fails, other processes can continue to serve inbound connections.

With integrated FastCGI spawning support, Supervisor gives you the best of both worlds. You get full-featured process management with groups of FastCGI processes sharing sockets without being tied to a particular web server. It's a clean separation of concerns, allowing the web server and the process manager to each do what they do best.

Note: The socket manager in Supervisor was originally developed to support FastCGI processes but it is not limited to FastCGI. Other protocols may be used as well with no special configuration. Any program that can access an open socket from a file descriptor (e.g. with socket.fromfd in Python) can use the socket manager. Supervisor will automatically create the socket, bind, and listen before forking the first child in a group. The socket will be passed to each child on file descriptor number `0` (zero). When the last child in the group exits, Supervisor will close the socket.

All the options available to `[program:x]` sections are also respected by `fcgi-program` sections.

[fcgi-program:x] Section Values

`[fcgi-program:x]` sections have a single key which `[program:x]` sections do not have.

`socket`

The FastCGI socket for this program, either TCP or UNIX domain socket. For TCP sockets, use this format: `tcp://localhost:9002`. For UNIX domain sockets, use `unix:///absolute/path/to/file.sock`. String expressions are evaluated against a dictionary containing the keys "program_name" and "here" (the directory of the supervisord config file).

Default: No default.

Required: Yes.

Introduced: 3.0

`socket_owner`

For UNIX domain sockets, this parameter can be used to specify the user and group for the FastCGI socket. May be a UNIX username (e.g. chrism) or a UNIX username and group separated by a colon (e.g. chrism:wheel).

Default: Uses the user and group set for the fcgi-program

Required: No.

Introduced: 3.0

`socket_mode`

For UNIX domain sockets, this parameter can be used to specify the permission mode.

Default: 0700

Required: No.

Introduced: 3.0

Consult *[program:x] Section Settings* for other allowable keys, delta the above constraints and additions.

`[fcgi-program:x]` Section Example

```
[fcgi-program:fcgiprogramname]
command=/usr/bin/example.fcgi
socket=unix:///var/run/supervisor/%(program_name)s.sock
socket_owner=chrism
socket_mode=0700
process_name=%(program_name)s_%(process_num)02d
numprocs=5
directory=/tmp
umask=022
priority=999
autostart=true
autorestart=unexpected
startsecs=1
startretries=3
exitcodes=0,2
stopsignal=QUIT
stopasgroup=false
killasgroup=false
stopwaitsecs=10
user=chrism
redirect_stderr=true
stdout_logfile=/a/path
stdout_logfile_maxbytes=1MB
stdout_logfile_backups=10
stdout_events_enabled=false
stderr_logfile=/a/path
stderr_logfile_maxbytes=1MB
stderr_logfile_backups=10
stderr_events_enabled=false
environment=A="1",B="2"
serverurl=AUTO
```

`[eventlistener:x]` Section Settings

Supervisor allows specialized homogeneous process groups ("event listener pools") to be defined within the configuration file. These pools contain processes that are meant to receive and respond to event notifications from supervisor's event system. See *Events* for an explanation of how events work and how to implement programs that can be declared as event listeners.

Note that all the options available to `[program:x]` sections are respected by eventlistener sections *except* for `stdout_capture_maxbytes`. Eventlisteners cannot emit process communication events on `stdout`, but can emit on `stderr` (see *Capture Mode*).

`[eventlistener:x]` Section Values

`[eventlistener:x]` sections have a few keys which `[program:x]` sections do not have.

`buffer_size`

> The event listener pool's event queue buffer size. When a listener pool's event buffer is overflowed (as can happen when an event listener pool cannot keep up with all of the events sent to it), the oldest event in the buffer is discarded.

`events`

> A comma-separated list of event type names that this listener is "interested" in receiving notifications for (see *Event Types* for a list of valid event type names).

`result_handler`

> A pkg_resources entry point string that resolves to a Python callable. The default value is `supervisor.dispatchers:default_handler`. Specifying an alternate result handler is a very uncommon thing to need to do, and as a result, how to create one is not documented.

Consult *[program:x] Section Settings* for other allowable keys, delta the above constraints and additions.

`[eventlistener:x]` Section Example

```
[eventlistener:theeventlistenername]
command=/bin/eventlistener
process_name=%(program_name)s_%(process_num)02d
numprocs=5
events=PROCESS_STATE
buffer_size=10
directory=/tmp
umask=022
priority=-1
autostart=true
autorestart=unexpected
startsecs=1
startretries=3
exitcodes=0,2
stopsignal=QUIT
stopwaitsecs=10
stopasgroup=false
killasgroup=false
user=chrism
redirect_stderr=false
stdout_logfile=/a/path
stdout_logfile_maxbytes=1MB
stdout_logfile_backups=10
stdout_events_enabled=false
stderr_logfile=/a/path
stderr_logfile_maxbytes=1MB
stderr_logfile_backups=10
stderr_events_enabled=false
```

```
environment=A="1",B="2"
serverurl=AUTO
```

[rpcinterface:x] Section Settings

Adding rpcinterface:x settings in the configuration file is only useful for people who wish to extend supervisor with additional custom behavior.

In the sample config file, there is a section which is named [rpcinterface:supervisor]. By default it looks like the following.

```
[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface
```

The [rpcinterface:supervisor] section *must* remain in the configuration for the standard setup of supervisor to work properly. If you don't want supervisor to do anything it doesn't already do out of the box, this is all you need to know about this type of section.

However, if you wish to add rpc interface namespaces in order to customize supervisor, you may add additional [rpcinterface:foo] sections, where "foo" represents the namespace of the interface (from the web root), and the value named by supervisor.rpcinterface_factory is a factory callable which should have a function signature that accepts a single positional argument supervisord and as many keyword arguments as required to perform configuration. Any extra key/value pairs defined within the [rpcinterface:x] section will be passed as keyword arguments to the factory.

Here's an example of a factory function, created in the __init__.py file of the Python package my.package.

```
from my.package.rpcinterface import AnotherRPCInterface

def make_another_rpcinterface(supervisord, **config):
    retries = int(config.get('retries', 0))
    another_rpc_interface = AnotherRPCInterface(supervisord, retries)
    return another_rpc_interface
```

And a section in the config file meant to configure it.

```
[rpcinterface:another]
supervisor.rpcinterface_factory = my.package:make_another_rpcinterface
retries = 1
```

[rpcinterface:x] Section Values

supervisor.rpcinterface_factory

pkg_resources "entry point" dotted name to your RPC interface's factory function.

Default: N/A

Required: No.

Introduced: 3.0

`[rpcinterface:x]` Section Example

```
[rpcinterface:another]
supervisor.rpcinterface_factory = my.package:make_another_rpcinterface
retries = 1
```

Subprocesses

supervisord's primary purpose is to create and manage processes based on data in its configuration file. It does this by creating subprocesses. Each subprocess spawned by supervisor is managed for the entirety of its lifetime by supervisord (**supervisord** is the parent process of each process it creates). When a child dies, supervisor is notified of its death via the `SIGCHLD` signal, and it performs the appropriate operation.

Nondaemonizing of Subprocesses

Programs meant to be run under supervisor should not daemonize themselves. Instead, they should run in the foreground. They should not detach from the terminal from which they are started.

The easiest way to tell if a program will run in the foreground is to run the command that invokes the program from a shell prompt. If it gives you control of the terminal back, but continues running, it's daemonizing itself and that will almost certainly be the wrong way to run it under supervisor. You want to run a command that essentially requires you to press `Ctrl-C` to get control of the terminal back. If it gives you a shell prompt back after running it without needing to press `Ctrl-C`, it's not useful under supervisor. All programs have options to be run in the foreground but there's no "standard way" to do it; you'll need to read the documentation for each program.

Below are configuration file examples that are known to start common programs in "foreground" mode under Supervisor.

Examples of Program Configurations

Here are some "real world" program configuration examples:

Apache 2.2.6

```
[program:apache2]
command=/path/to/httpd -c "ErrorLog /dev/stdout" -DFOREGROUND
redirect_stderr=true
```

Two Zope 2.X instances and one ZEO server

```
[program:zeo]
command=/path/to/runzeo
priority=1

[program:zope1]
command=/path/to/instance/home/bin/runzope
priority=2
redirect_stderr=true
```

```
[program:zope2]
command=/path/to/another/instance/home/bin/runzope
priority=2
redirect_stderr=true
```

Postgres 8.X

```
[program:postgres]
command=/path/to/postmaster
; we use the "fast" shutdown signal SIGINT
stopsignal=INT
redirect_stderr=true
```

OpenLDAP `slapd`

```
[program:slapd]
command=/path/to/slapd -f /path/to/slapd.conf -h ldap://0.0.0.0:8888
redirect_stderr=true
```

Other Examples

Other examples of shell scripts that could be used to start services under **supervisord** can be found at http://thedjbway.b0llix.net/services.html. These examples are actually for **daemontools** but the premise is the same for supervisor.

Another collection of recipes for starting various programs in the foreground is available from http://smarden.org/runit/runscripts.html.

`pidproxy` Program

Some processes (like **mysqld**) ignore signals sent to the actual process which is spawned by **supervisord**. Instead, a "special" thread/process is created by these kinds of programs which is responsible for handling signals. This is problematic because **supervisord** can only kill a process which it creates itself. If a process created by **supervisord** creates its own child processes, **supervisord** cannot kill them.

Fortunately, these types of programs typically write a "pidfile" which contains the "special" process' PID, and is meant to be read and used in order to kill the process. As a workaround for this case, a special **pidproxy** program can handle startup of these kinds of processes. The **pidproxy** program is a small shim that starts a process, and upon the receipt of a signal, sends the signal to the pid provided in a pidfile. A sample configuration program entry for a pidproxy-enabled program is provided below.

```
[program:mysql]
command=/path/to/pidproxy /path/to/pidfile /path/to/mysqld_safe
```

The **pidproxy** program is put into your configuration's `$BINDIR` when supervisor is installed (it is a "console script").

Subprocess Environment

Subprocesses will inherit the environment of the shell used to start the **supervisord** program. Several environment variables will be set by **supervisord** itself in the child's environment also, including SUPERVISOR_ENABLED (a flag indicating the process is under supervisor control), SUPERVISOR_PROCESS_NAME (the config-file-specified process name for this process) and SUPERVISOR_GROUP_NAME (the config-file-specified process group name for the child process).

These environment variables may be overridden within the [supervisord] section config option named environment (applies to all subprocesses) or within the per- [program:x] section environment config option (applies only to the subprocess specified within the [program:x] section). These "environment" settings are additive. In other words, each subprocess' environment will consist of:

The environment variables set within the shell used to start supervisord...

... added-to/overridden-by ...

... the environment variables set within the "environment" global

config option ...

... added-to/overridden-by ...

... supervisor-specific environment variables (SUPERVISOR_ENABLED, SUPERVISOR_PROCESS_NAME, SUPERVISOR_GROUP_NAME) ..

... added-to/overridden-by ...

... the environment variables set within the per-process "environment" config option.

No shell is executed by **supervisord** when it runs a subprocess, so environment variables such as USER, PATH, HOME, SHELL, LOGNAME, etc. are not changed from their defaults or otherwise reassigned. This is particularly important to note when you are running a program from a **supervisord** run as root with a user= stanza in the configuration. Unlike **cron**, **supervisord** does not attempt to divine and override "fundamental" environment variables like USER, PATH, HOME, and LOGNAME when it performs a setuid to the user defined within the user= program config option. If you need to set environment variables for a particular program that might otherwise be set by a shell invocation for a particular user, you must do it explicitly within the environment= program config option. An example of setting these environment variables is as below.

```
[program:apache2]
command=/home/chrism/bin/httpd -c "ErrorLog /dev/stdout" -DFOREGROUND
user=chrism
environment=HOME="/home/chrism",USER="chrism"
```

Process States

A process controlled by supervisord will be in one of the below states at any given time. You may see these state names in various user interface elements in clients.

STOPPED (0)

The process has been stopped due to a stop request or has never been started.

STARTING (10)

The process is starting due to a start request.

RUNNING (20)

The process is running.

BACKOFF (30)

The process entered the STARTING state but subsequently exited too quickly to move to the RUNNING state.

STOPPING (40)

The process is stopping due to a stop request.

EXITED (100)

The process exited from the RUNNING state (expectedly or unexpectedly).

FATAL (200)

The process could not be started successfully.

UNKNOWN (1000)

The process is in an unknown state (**supervisord** programming error).

Each process run under supervisor progresses through these states as per the following directed graph.

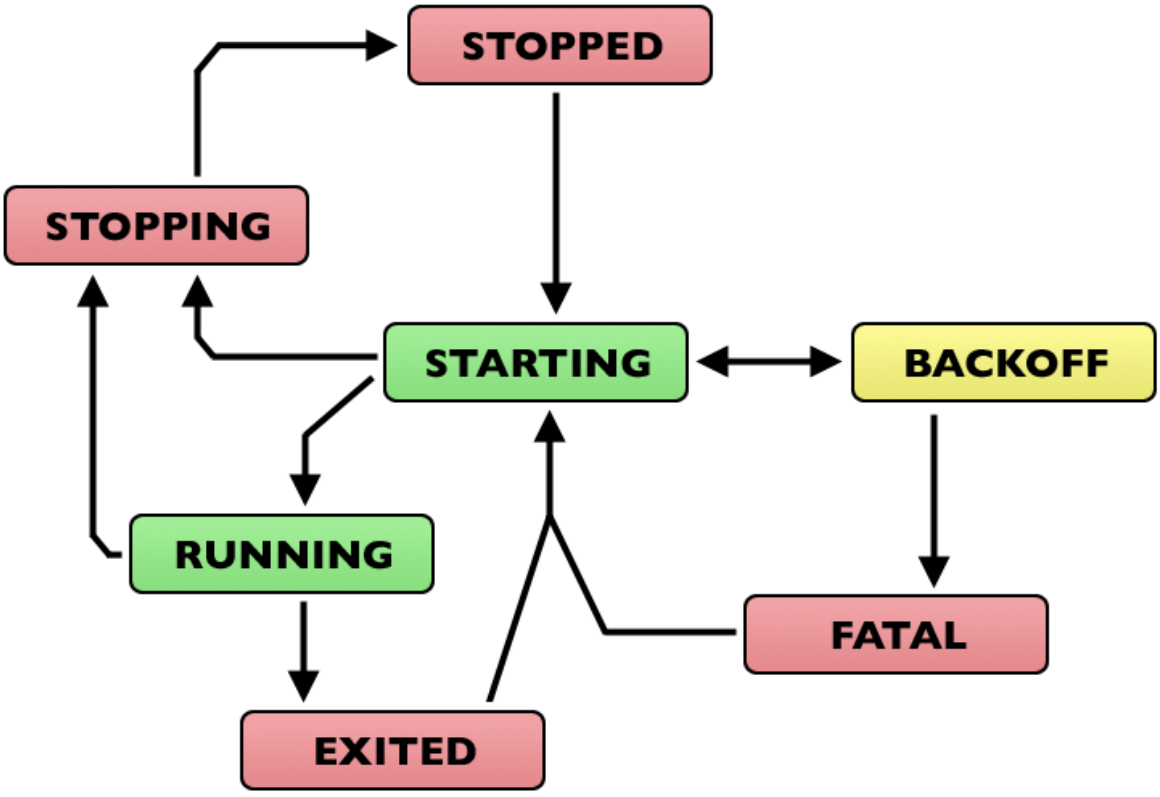

Fig. 1.1: Subprocess State Transition Graph

A process is in the STOPPED state if it has been stopped adminstratively or if it has never been started.

When an autorestarting process is in the BACKOFF state, it will be automatically restarted by **supervisord**. It will switch between STARTING and BACKOFF states until it becomes evident that it cannot be started because the number of startretries has exceeded the maximum, at which point it will transition to the FATAL state. Each start retry will take progressively more time.

When a process is in the EXITED state, it will automatically restart:

- never if its `autorestart` parameter is set to `false`.
- unconditionally if its `autorestart` parameter is set to `true`.
- conditionally if its `autorestart` parameter is set to `unexpected`. If it exited with an exit code that doesn't match one of the exit codes defined in the `exitcodes` configuration parameter for the process, it will be restarted.

A process automatically transitions from `EXITED` to `RUNNING` as a result of being configured to autorestart conditionally or unconditionally. The number of transitions between `RUNNING` and `EXITED` is not limited in any way: it is possible to create a configuration that endlessly restarts an exited process. This is a feature, not a bug.

An autorestarted process will never be automatically restarted if it ends up in the `FATAL` state (it must be manually restarted from this state).

A process transitions into the `STOPPING` state via an administrative stop request, and will then end up in the `STOPPED` state.

A process that cannot be stopped successfully will stay in the `STOPPING` state forever. This situation should never be reached during normal operations as it implies that the process did not respond to a final `SIGKILL` signal sent to it by supervisor, which is "impossible" under UNIX.

State transitions which always require user action to invoke are these:

`FATAL` -> `STARTING`

`RUNNING` -> `STOPPING`

State transitions which typically, but not always, require user action to invoke are these, with exceptions noted:

`STOPPED` -> `STARTING` (except at supervisord startup if process is configured to autostart)

`EXITED` -> `STARTING` (except if process is configured to autorestart)

All other state transitions are managed by supervisord automatically.

Logging

One of the main tasks that **`supervisord`** performs is logging. **`supervisord`** logs an activity log detailing what it's doing as it runs. It also logs child process stdout and stderr output to other files if configured to do so.

Activity Log

The activity log is the place where **`supervisord`** logs messages about its own health, its subprocess' state changes, any messages that result from events, and debug and informational messages. The path to the activity log is configured via the `logfile` parameter in the `[supervisord]` section of the configuration file, defaulting to `$CWD/supervisord.log`. Sample activity log traffic is shown in the example below. Some lines have been broken to better fit the screen.

Sample Activity Log Output

```
2007-09-08 14:43:22,886 DEBG 127.0.0.1:Medusa (V1.11) started at Sat Sep  8 14:43:22␣
↪2007
        Hostname: kingfish
        Port:9001
2007-09-08 14:43:22,961 INFO RPC interface 'supervisor' initialized
2007-09-08 14:43:22,961 CRIT Running without any HTTP authentication checking
```

```
2007-09-08 14:43:22,962 INFO supervisord started with pid 27347
2007-09-08 14:43:23,965 INFO spawned: 'listener_00' with pid 27349
2007-09-08 14:43:23,970 INFO spawned: 'eventgen' with pid 27350
2007-09-08 14:43:23,990 INFO spawned: 'grower' with pid 27351
2007-09-08 14:43:24,059 DEBG 'listener_00' stderr output:
 /Users/chrism/projects/supervisor/supervisor2/dev-sandbox/bin/python:
 can't open file '/Users/chrism/projects/supervisor/supervisor2/src/supervisor/
↪scripts/osx_eventgen_listener.py':
 [Errno 2] No such file or directory
2007-09-08 14:43:24,060 DEBG fd 7 closed, stopped monitoring
↪<PEventListenerDispatcher at 19910168 for
 <Subprocess at 18892960 with name listener_00 in state STARTING> (stdout)>
2007-09-08 14:43:24,060 INFO exited: listener_00 (exit status 2; not expected)
2007-09-08 14:43:24,061 DEBG received SIGCHLD indicating a child quit
```

The activity log "level" is configured in the config file via the `loglevel` parameter in the `[supervisord]` ini file section. When `loglevel` is set, messages of the specified priority, plus those with any higher priority are logged to the activity log. For example, if `loglevel` is `error`, messages of `error` and `critical` priority will be logged. However, if loglevel is `warn`, messages of `warn`, `error`, and `critical` will be logged.

Activity Log Levels

The below table describes the logging levels in more detail, ordered in highest priority to lowest. The "Config File Value" is the string provided to the `loglevel` parameter in the `[supervisord]` section of configuration file and the "Output Code" is the code that shows up in activity log output lines.

Config File Value	Output Code	Description
critical	CRIT	Messages that indicate a condition that requires immediate user attention, a supervisor state change, or an error in supervisor itself.
error	ERRO	Messages that indicate a potentially ignorable error condition (e.g. unable to clear a log directory).
warn	WARN	Messages that indicate an anomalous condition which isn't an error.
info	INFO	Normal informational output. This is the default log level if none is explicitly configured.
debug	DEBG	Messages useful for users trying to debug process configuration and communications behavior (process output, listener state changes, event notifications).
trace	TRAC	Messages useful for developers trying to debug supervisor plugins, and information about HTTP and RPC requests and responses.
blather	BLAT	Messages useful for developers trying to debug supervisor itself.

Activity Log Rotation

The activity log is "rotated" by **`supervisord`** based on the combination of the `logfile_maxbytes` and the `logfile_backups` parameters in the `[supervisord]` section of the configuration file. When the activity log reaches `logfile_maxbytes` bytes, the current log file is moved to a backup file and a new activity log file is created. When this happens, if the number of existing backup files is greater than or equal to `logfile_backups`, the oldest backup file is removed and the backup files are renamed accordingly. If the file being written to is named `supervisord.log`, when it exceeds `logfile_maxbytes`, it is closed and renamed to `supervisord.log.1`, and if files `supervisord.log.1`, `supervisord.log.2` etc. exist, then they are renamed to `supervisord.log.2`, `supervisord.log.3` etc. respectively. If `logfile_maxbytes` is 0, the logfile is never rotated (and thus backups are never made). If `logfile_backups` is 0, no backups will be kept.

Child Process Logs

The stdout of child processes spawned by supervisor, by default, is captured for redisplay to users of **supervisorctl** and other clients. If no specific logfile-related configuration is performed in a `[program:x]`, `[fcgi-program:x]`, or `[eventlistener:x]` section in the configuration file, the following is true:

- **supervisord** will capture the child process' stdout and stderr output into temporary files. Each stream is captured to a separate file. This is known as `AUTO` log mode.
- `AUTO` log files are named automatically and placed in the directory configured as `childlogdir` of the `[supervisord]` section of the config file.
- The size of each `AUTO` log file is bounded by the `{streamname}_logfile_maxbytes` value of the program section (where {streamname} is "stdout" or "stderr"). When it reaches that number, it is rotated (like the activity log), based on the `{streamname}_logfile_backups`.

The configuration keys that influence child process logging in `[program:x]` and `[fcgi-program:x]` sections are these:

`redirect_stderr`, `stdout_logfile`, `stdout_logfile_maxbytes`, `stdout_logfile_backups`, `stdout_capture_maxbytes`, `stdout_syslog`, `stderr_logfile`, `stderr_logfile_maxbytes`, `stderr_logfile_backups`, `stderr_capture_maxbytes`, and `stderr_syslog`.

`[eventlistener:x]` sections may not specify `redirect_stderr`, `stdout_capture_maxbytes`, or `stderr_capture_maxbytes`, but otherwise they accept the same values.

The configuration keys that influence child process logging in the `[supervisord]` config file section are these: `childlogdir`, and `nocleanup`.

Capture Mode

Capture mode is an advanced feature of Supervisor. You needn't understand capture mode unless you want to take actions based on data parsed from subprocess output.

If a `[program:x]` section in the configuration file defines a non-zero `stdout_capture_maxbytes` or `stderr_capture_maxbytes` parameter, each process represented by the program section may emit special tokens on its stdout or stderr stream (respectively) which will effectively cause supervisor to emit a `PROCESS_COMMUNICATION` event (see *Events* for a description of events).

The process communications protocol relies on two tags, one which commands supervisor to enter "capture mode" for the stream and one which commands it to exit. When a process stream enters "capture mode", data sent to the stream will be sent to a separate buffer in memory, the "capture buffer", which is allowed to contain a maximum of `capture_maxbytes` bytes. During capture mode, when the buffer's length exceeds `capture_maxbytes` bytes, the earliest data in the buffer is discarded to make room for new data. When a process stream exits capture mode, a `PROCESS_COMMUNICATION` event subtype is emitted by supervisor, which may be intercepted by event listeners.

The tag to begin "capture mode" in a process stream is `<!--XSUPERVISOR:BEGIN-->`. The tag to exit capture mode is `<!--XSUPERVISOR:END-->`. The data between these tags may be arbitrary, and forms the payload of the `PROCESS_COMMUNICATION` event. For example, if a program is set up with a `stdout_capture_maxbytes` of "1MB", and it emits the following on its stdout stream:

```
<!--XSUPERVISOR:BEGIN-->Hello!<!--XSUPERVISOR:END-->
```

In this circumstance, **supervisord** will emit a `PROCESS_COMMUNICATIONS_STDOUT` event with data in the payload of "Hello!".

An example of a script (written in Python) which emits a process communication event is in the `scripts` directory of the supervisor package, named `sample_commevent.py`.

The output of processes specified as "event listeners" (`[eventlistener:x]` sections) is not processed this way. Output from these processes cannot enter capture mode.

Events

Events are an advanced feature of Supervisor introduced in version 3.0. You don't need to understand events if you simply want to use Supervisor as a mechanism to restart crashed processes or as a system to manually control process state. You do need to understand events if you want to use Supervisor as part of a process monitoring/notification framework.

Event Listeners and Event Notifications

Supervisor provides a way for a specially written program (which it runs as a subprocess) called an "event listener" to subscribe to "event notifications". An event notification implies that something happened related to a subprocess controlled by **supervisord** or to **supervisord** itself. Event notifications are grouped into types in order to make it possible for event listeners to subscribe to a limited subset of event notifications. Supervisor continually emits event notifications as its running even if there are no listeners configured. If a listener is configured and subscribed to an event type that is emitted during a **supervisord** lifetime, that listener will be notified.

The purpose of the event notification/subscription system is to provide a mechanism for arbitrary code to be run (e.g. send an email, make an HTTP request, etc) when some condition is met. That condition usually has to do with subprocess state. For instance, you may want to notify someone via email when a process crashes and is restarted by Supervisor.

The event notification protocol is based on communication via a subprocess' stdin and stdout. Supervisor sends specially-formatted input to an event listener process' stdin and expects specially-formatted output from an event listener's stdout, forming a request-response cycle. A protocol agreed upon between supervisor and the listener's implementer allows listeners to process event notifications. Event listeners can be written in any language supported by the platform you're using to run Supervisor. Although event listeners may be written in any language, there is special library support for Python in the form of a `supervisor.childutils` module, which makes creating event listeners in Python slightly easier than in other languages.

Configuring an Event Listener

A supervisor event listener is specified via a `[eventlistener:x]` section in the configuration file. Supervisor `[eventlistener:x]` sections are treated almost exactly like supervisor `[program:x]` section with the respect to the keys allowed in their configuration except that Supervisor does not respect "capture mode" output from event listener processes (ie. event listeners cannot be `PROCESS_COMMUNICATIONS_EVENT` event generators). Therefore it is an error to specify `stdout_capture_maxbytes` or `stderr_capture_maxbytes` in the configuration of an eventlistener. There is no artificial constraint on the number of eventlistener sections that can be placed into the configuration file.

When an `[eventlistener:x]` section is defined, it actually defines a "pool", where the number of event listeners in the pool is determined by the `numprocs` value within the section.

The `events` parameter of the `[eventlistener:x]` section specifies the events that will be sent to a listener pool. A well-written event listener will ignore events that it cannot process, but there is no guarantee that a specific event listener won't crash as a result of receiving an event type it cannot handle. Therefore, depending on the listener implementation, it may be important to specify in the configuration that it may receive only certain types of events. The implementor of the event listener is the only person who can tell you what these are (and therefore what value to put in the `events` configuration). Examples of eventlistener configurations that can be placed in `supervisord.conf` are as follows.

```
[eventlistener:memmon]
command=memmon -a 200MB -m bob@example.com
events=TICK_60
```

```
[eventlistener:mylistener]
command=my_custom_listener.py
events=PROCESS_STATE,TICK_60
```

Note: An advanced feature, specifying an alternate "result handler" for a pool, can be specified via the `result_handler` parameter of an `[eventlistener:x]` section in the form of a pkg_resources "entry point" string. The default result handler is `supervisord.dispatchers:default_handler`. Creating an alternate result handler is not currently documented.

When an event notification is sent by supervisor, all event listener pools which are subscribed to receive events for the event's type (filtered by the `events` value in the eventlistener section) will be found. One of the listeners in each listener pool will receive the event notification (any "available" listener).

Every process in an event listener pool is treated equally by supervisor. If a process in the pool is unavailable (because it is already processing an event, because it has crashed, or because it has elected to removed itself from the pool), supervisor will choose another process from the pool. If the event cannot be sent because all listeners in the pool are "busy", the event will be buffered and notification will be retried later. "Later" is defined as "the next time that the **supervisord** select loop executes". For satisfactory event processing performance, you should configure a pool with as many event listener processes as appropriate to handle your event load. This can only be determined empirically for any given workload, there is no "magic number" but to help you determine the optimal number of listeners in a given pool, Supervisor will emit warning messages to its activity log when an event cannot be sent immediately due to pool congestion. There is no artificial constraint placed on the number of processes that can be in a pool, it is limited only by your platform constraints.

A listener pool has an event buffer queue. The queue is sized via the listener pool's `buffer_size` config file option. If the queue is full and supervisor attempts to buffer an event, supervisor will throw away the oldest event in the buffer and log an error.

Writing an Event Listener

An event listener implementation is a program that is willing to accept structured input on its stdin stream and produce structured output on its stdout stream. An event listener implementation should operate in "unbuffered" mode or should flush its stdout every time it needs to communicate back to the supervisord process. Event listeners can be written to be long-running or may exit after a single request (depending on the implementation and the `autorestart` parameter in the eventlistener's configuration).

An event listener can send arbitrary output to its stderr, which will be logged or ignored by supervisord depending on the stderr-related logfile configuration in its `[eventlistener:x]` section.

Event Notification Protocol

When supervisord sends a notification to an event listener process, the listener will first be sent a single "header" line on its stdin. The composition of the line is a set of colon-separated tokens (each of which represents a key-value pair) separated from each other by a single space. The line is terminated with a `\n` (linefeed) character. The tokens on the line are not guaranteed to be in any particular order. The types of tokens currently defined are in the table below.

Header Tokens

Key	Description	Example
ver	The event system protocol version	3.0
server	The identifier of the supervisord sending the event (see config file `[supervisord]` section `identifier` value.	
serial	An integer assigned to each event. No two events generated during the lifetime of a **`supervisord`** process will have the same serial number. The value is useful for functional testing and detecting event ordering anomalies.	30
pool	The name of the event listener pool which generated this event.	myeventpool
poolserial	An integer assigned to each event by the eventlistener pool which it is being sent from. No two events generated by the same eventlister pool during the lifetime of a **`supervisord`** process will have the same `poolserial` number. This value can be used to detect event ordering anomalies.	30
eventname	The specific event type name (see *Event Types*)	TICK_5
len	An integer indicating the number of bytes in the event payload, aka the `PAYLOAD_LENGTH`	22

An example of a complete header line is as follows.

```
ver:3.0 server:supervisor serial:21 pool:listener poolserial:10 eventname:PROCESS_
↪COMMUNICATION_STDOUT len:54
```

Directly following the linefeed character in the header is the event payload. It consists of `PAYLOAD_LENGTH` bytes representing a serialization of the event data. See *Event Types* for the specific event data serialization definitions.

An example payload for a `PROCESS_COMMUNICATION_STDOUT` event notification is as follows.

```
processname:foo groupname:bar pid:123
This is the data that was sent between the tags
```

The payload structure of any given event is determined only by the event's type.

Event Listener States

An event listener process has three possible states that are maintained by supervisord:

Name	Description
ACKNOWLEDGED	The event listener has acknowledged (accepted or rejected) an event send.
READY	Event notificatons may be sent to this event listener
BUSY	Event notifications may not be sent to this event listener.

When an event listener process first starts, supervisor automatically places it into the `ACKNOWLEDGED` state to allow for startup activities or guard against startup failures (hangs). Until the listener sends a `READY\n` string to its stdout, it will stay in this state.

When supervisor sends an event notification to a listener in the `READY` state, the listener will be placed into the `BUSY` state until it receives an `OK` or `FAIL` response from the listener, at which time, the listener will be transitioned back into the `ACKNOWLEDGED` state.

Event Listener Notification Protocol

Supervisor will notify an event listener in the `READY` state of an event by sending data to the stdin of the process. Supervisor will never send anything to the stdin of an event listener process while that process is in the `BUSY` or `ACKNOWLEDGED` state. Supervisor starts by sending the header.

Once it has processed the header, the event listener implementation should read `PAYLOAD_LENGTH` bytes from its stdin, perform an arbitrary action based on the values in the header and the data parsed out of the serialization. It is free to block for an arbitrary amount of time while doing this. Supervisor will continue processing normally as it waits for a response and it will send other events of the same type to other listener processes in the same pool as necessary.

After the event listener has processed the event serialization, in order to notify supervisord about the result, it should send back a result structure on its stdout. A result structure is the word "RESULT", followed by a space, followed by the result length, followed by a line feed, followed by the result content. For example, `RESULT 2\nOK` is the result "OK". Conventionally, an event listener will use either `OK` or `FAIL` as the result content. These strings have special meaning to the default result handler.

If the default result handler receives `OK` as result content, it will assume that the listener processed the event notification successfully. If it receives `FAIL`, it will assume that the listener has failed to process the event, and the event will be rebuffered and sent again at a later time. The event listener may reject the event for any reason by returning a `FAIL` result. This does not indicate a problem with the event data or the event listener. Once an `OK` or `FAIL` result is received by supervisord, the event listener is placed into the `ACKNOWLEDGED` state.

Once the listener is in the `ACKNOWLEDGED` state, it may either exit (and subsequently may be restarted by supervisor if its `autorestart` config parameter is `true`), or it may continue running. If it continues to run, in order to be placed back into the `READY` state by supervisord, it must send a `READY` token followed immediately by a line feed to its stdout.

Example Event Listener Implementation

A Python implementation of a "long-running" event listener which accepts an event notification, prints the header and payload to its stderr, and responds with an `OK` result, and then subsequently a `READY` is as follows.

```
import sys

def write_stdout(s):
    # only eventlistener protocol messages may be sent to stdout
    sys.stdout.write(s)
    sys.stdout.flush()

def write_stderr(s):
    sys.stderr.write(s)
    sys.stderr.flush()

def main():
    while 1:
        # transition from ACKNOWLEDGED to READY
        write_stdout('READY\n')

        # read header line and print it to stderr
        line = sys.stdin.readline()
        write_stderr(line)

        # read event payload and print it to stderr
        headers = dict([ x.split(':') for x in line.split() ])
        data = sys.stdin.read(int(headers['len']))
        write_stderr(data)
```

```
        # transition from READY to ACKNOWLEDGED
        write_stdout('RESULT 2\nOK')

if __name__ == '__main__':
    main()
```

Other sample event listeners are present within the *superlance* package, including one which can monitor supervisor subprocesses and restart a process if it is using "too much" memory.

Event Listener Error Conditions

If the event listener process dies while the event is being transmitted to its stdin, or if it dies before sending an result structure back to supervisord, the event is assumed to not be processed and will be rebuffered by supervisord and sent again later.

If an event listener sends data to its stdout which supervisor does not recognize as an appropriate response based on the state that the event listener is in, the event listener will be placed into the `UNKNOWN` state, and no further event notifications will be sent to it. If an event was being processed by the listener during this time, it will be rebuffered and sent again later.

Miscellaneous

Event listeners may use the Supervisor XML-RPC interface to call "back in" to Supervisor. As such, event listeners can impact the state of a Supervisor subprocess as a result of receiving an event notification. For example, you may want to generate an event every few minutes related to process usage of Supervisor-controlled subprocesses, and if any of those processes exceed some memory threshold, you would like to restart it. You would write a program that caused supervisor to generate `PROCESS_COMMUNICATION` events every so often with memory information in them, and an event listener to perform an action based on processing the data it receives from these events.

Event Types

The event types are a controlled set, defined by Supervisor itself. There is no way to add an event type without changing **`supervisord`** itself. This is typically not a problem, though, because metadata is attached to events that can be used by event listeners as additional filter criterion, in conjunction with its type.

Event types that may be subscribed to by event listeners are predefined by supervisor and fall into several major categories, including "process state change", "process communication", and "supervisor state change" events. Below are tables describing these event types.

In the below list, we indicate that some event types have a "body" which is a a *token set*. A token set consists of a set of charaters with space-separated tokens. Each token represents a key-value pair. The key and value are separated by a colon. For example:

```
processname:cat groupname:cat from_state:STOPPED
```

Token sets do not have a linefeed or carriage return character at their end.

`EVENT` Event Type

The base event type. This event type is abstract. It will never be sent directly. Subscribing to this event type will cause a subscriber to receive all event notifications emitted by Supervisor.

Name: EVENT

Subtype Of: N/A

Body Description: N/A

PROCESS_STATE Event Type

This process type indicates a process has moved from one state to another. See *Process States* for a description of the states that a process moves through during its lifetime. This event type is abstract, it will never be sent directly. Subscribing to this event type will cause a subscriber to receive event notifications of all the event types that are subtypes of PROCESS_STATE.

Name: PROCESS_STATE

Subtype Of: EVENT

Body Description

All subtypes of PROCESS_STATE have a body which is a token set. Additionally, each PROCESS_STATE subtype's token set has a default set of key/value pairs: processname, groupname, and from_state. processname represents the process name which supervisor knows this process as. groupname represents the name of the supervisord group which this process is in. from_state is the name of the state from which this process is transitioning (the new state is implied by the concrete event type). Concrete subtypes may include additional key/value pairs in the token set.

PROCESS_STATE_STARTING Event Type

Indicates a process has moved from a state to the STARTING state.

Name: PROCESS_STATE_STARTING

Subtype Of: PROCESS_STATE

Body Description

This body is a token set. It has the default set of key/value pairs plus an additional tries key. tries represents the number of times this process has entered this state before transitioning to RUNNING or FATAL (it will never be larger than the "startretries" parameter of the process). For example:

```
processname:cat groupname:cat from_state:STOPPED tries:0
```

PROCESS_STATE_RUNNING Event Type

Indicates a process has moved from the STARTING state to the RUNNING state. This means that the process has successfully started as far as Supervisor is concerned.

Name: PROCESS_STATE_RUNNING

Subtype Of: PROCESS_STATE

Body Description

This body is a token set. It has the default set of key/value pairs plus an additional `pid` key. `pid` represents the UNIX process id of the process that was started. For example:

```
processname:cat groupname:cat from_state:STARTING pid:2766
```

`PROCESS_STATE_BACKOFF` Event Type

Indicates a process has moved from the `STARTING` state to the `BACKOFF` state. This means that the process did not successfully enter the RUNNING state, and Supervisor is going to try to restart it unless it has exceeded its "startretries" configuration limit.

Name: `PROCESS_STATE_BACKOFF`

Subtype Of: `PROCESS_STATE`

Body Description

This body is a token set. It has the default set of key/value pairs plus an additional `tries` key. `tries` represents the number of times this process has entered this state before transitioning to `RUNNING` or `FATAL` (it will never be larger than the "startretries" parameter of the process). For example:

```
processname:cat groupname:cat from_state:STOPPED tries:0
```

`PROCESS_STATE_STOPPING` Event Type

Indicates a process has moved from either the `RUNNING` state or the `STARTING` state to the `STOPPING` state.

Name: `PROCESS_STATE_STOPPING`

Subtype Of: `PROCESS_STATE`

Body Description

This body is a token set. It has the default set of key/value pairs plus an additional `pid` key. `pid` represents the UNIX process id of the process that was started. For example:

```
processname:cat groupname:cat from_state:STARTING pid:2766
```

`PROCESS_STATE_EXITED` Event Type

Indicates a process has moved from the `RUNNING` state to the `EXITED` state.

Name: `PROCESS_STATE_EXITED`

Subtype Of: `PROCESS_STATE`

Body Description

This body is a token set. It has the default set of key/value pairs plus two additional keys: `pid` and `expected`. `pid` represents the UNIX process id of the process that exited. `expected` represents whether the process exited with an expected exit code or not. It will be `0` if the exit code was unexpected, or `1` if the exit code was expected. For example:

```
processname:cat groupname:cat from_state:RUNNING expected:0 pid:2766
```

PROCESS_STATE_STOPPED Event Type

Indicates a process has moved from the `STOPPING` state to the `STOPPED` state.

Name: `PROCESS_STATE_STOPPED`

Subtype Of: `PROCESS_STATE`

Body Description

This body is a token set. It has the default set of key/value pairs plus an additional `pid` key. `pid` represents the UNIX process id of the process that was started. For example:

```
processname:cat groupname:cat from_state:STOPPING pid:2766
```

PROCESS_STATE_FATAL Event Type

Indicates a process has moved from the `BACKOFF` state to the `FATAL` state. This means that Supervisor tried `startretries` number of times unsuccessfully to start the process, and gave up attempting to restart it.

Name: `PROCESS_STATE_FATAL`

Subtype Of: `PROCESS_STATE`

Body Description

This event type is a token set with the default key/value pairs. For example:

```
processname:cat groupname:cat from_state:BACKOFF
```

PROCESS_STATE_UNKNOWN Event Type

Indicates a process has moved from any state to the `UNKNOWN` state (indicates an error in **`supervisord`**). This state transition will only happen if **`supervisord`** itself has a programming error.

Name: `PROCESS_STATE_UNKNOWN`

Subtype Of: `PROCESS_STATE`

Body Description

This event type is a token set with the default key/value pairs. For example:

```
processname:cat groupname:cat from_state:BACKOFF
```

REMOTE_COMMUNICATION Event Type

An event type raised when the `supervisor.sendRemoteCommEvent()` method is called on Supervisor's RPC interface. The `type` and `data` are arguments of the RPC method.

Name: `REMOTE_COMMUNICATION`

Subtype Of: `EVENT`

Body Description

```
type:type
data
```

PROCESS_LOG Event Type

An event type emitted when a process writes to stdout or stderr. The event will only be emitted if the file descriptor is not in capture mode and if `stdout_events_enabled` or `stderr_events_enabled` config options are set to `true`. This event type is abstract, it will never be sent directly. Subscribing to this event type will cause a subscriber to receive event notifications for all subtypes of `PROCESS_LOG`.

Name: `PROCESS_LOG`

Subtype Of: `EVENT`

Body Description: N/A

PROCESS_LOG_STDOUT Event Type

Indicates a process has written to its stdout file descriptor. The event will only be emitted if the file descriptor is not in capture mode and if the `stdout_events_enabled` config option is set to `true`.

Name: `PROCESS_LOG_STDOUT`

Subtype Of: `PROCESS_LOG`

Body Description

```
processname:name groupname:name pid:pid
data
```

PROCESS_LOG_STDERR Event Type

Indicates a process has written to its stderr file descriptor. The event will only be emitted if the file descriptor is not in capture mode and if the `stderr_events_enabled` config option is set to `true`.

Name: `PROCESS_LOG_STDERR`

Subtype Of: `PROCESS_LOG`

Body Description

```
processname:name groupname:name pid:pid
data
```

PROCESS_COMMUNICATION Event Type

An event type raised when any process attempts to send information between `<!--XSUPERVISOR:BEGIN-->` and `<!--XSUPERVISOR:END-->` tags in its output. This event type is abstract, it will never be sent directly. Subscribing to this event type will cause a subscriber to receive event notifications for all subtypes of `PROCESS_COMMUNICATION`.

Name: `PROCESS_COMMUNICATION`

Subtype Of: `EVENT`

Body Description: N/A

PROCESS_COMMUNICATION_STDOUT Event Type

Indicates a process has sent a message to Supervisor on its stdout file descriptor.

Name: `PROCESS_COMMUNICATION_STDOUT`

Subtype Of: `PROCESS_COMMUNICATION`

Body Description

```
processname:name groupname:name pid:pid
data
```

PROCESS_COMMUNICATION_STDERR Event Type

Indicates a process has sent a message to Supervisor on its stderr file descriptor.

Name: `PROCESS_COMMUNICATION_STDERR`

Subtype Of: `PROCESS_COMMUNICATION`

Body Description

```
processname:name groupname:name pid:pid
data
```

SUPERVISOR_STATE_CHANGE Event Type

An event type raised when the state of the **supervisord** process changes. This type is abstract, it will never be sent directly. Subscribing to this event type will cause a subscriber to receive event notifications of all the subtypes of SUPERVISOR_STATE_CHANGE.

Name: SUPERVISOR_STATE_CHANGE

Subtype Of: EVENT

Body Description: N/A

SUPERVISOR_STATE_CHANGE_RUNNING Event Type

Indicates that **supervisord** has started.

Name: SUPERVISOR_STATE_CHANGE_RUNNING

Subtype Of: SUPERVISOR_STATE_CHANGE

Body Description: Empty string

SUPERVISOR_STATE_CHANGE_STOPPING Event Type

Indicates that **supervisord** is stopping.

Name: SUPERVISOR_STATE_CHANGE_STOPPING

Subtype Of: SUPERVISOR_STATE_CHANGE

Body Description: Empty string

TICK Event Type

An event type that may be subscribed to for event listeners to receive "wake-up" notifications every N seconds. This event type is abstract, it will never be sent directly. Subscribing to this event type will cause a subscriber to receive event notifications for all subtypes of TICK.

Note that the only TICK events available are the ones listed below. You cannot subscribe to an arbitrary TICK interval. If you need an interval not provided below, you can subscribe to one of the shorter intervals given below and keep track of the time between runs in your event listener.

Name: TICK

Subtype Of: EVENT

Body Description: N/A

TICK_5 Event Type

An event type that may be subscribed to for event listeners to receive "wake-up" notifications every 5 seconds.

Name: TICK_5

Subtype Of: TICK

Body Description

This event type is a token set with a single key: "when", which indicates the epoch time for which the tick was sent.

```
when:1201063880
```

`TICK_60` Event Type

An event type that may be subscribed to for event listeners to receive "wake-up" notifications every 60 seconds.

Name: `TICK_60`

Subtype Of: `TICK`

Body Description

This event type is a token set with a single key: "when", which indicates the epoch time for which the tick was sent.

```
when:1201063880
```

`TICK_3600` Event Type

An event type that may be subscribed to for event listeners to receive "wake-up" notifications every 3600 seconds (1 hour).

Name: `TICK_3600`

Subtype Of: `TICK`

Body Description

This event type is a token set with a single key: "when", which indicates the epoch time for which the tick was sent.

```
when:1201063880
```

`PROCESS_GROUP` Event Type

An event type raised when a process group is added to or removed from Supervisor. This type is abstract, it will never be sent directly. Subscribing to this event type will cause a subscriber to receive event notifications of all the subtypes of `PROCESS_GROUP`.

Name: `PROCESS_GROUP`

Subtype Of: `EVENT`

Body Description: N/A

PROCESS_GROUP_ADDED Event Type

Indicates that a process group has been added to Supervisor's configuration.

Name: PROCESS_GROUP_ADDED

Subtype Of: PROCESS_GROUP

Body Description: This body is a token set with just a groupname key/value.

```
groupname:cat
```

PROCESS_GROUP_REMOVED Event Type

Indicates that a process group has been removed from Supervisor's configuration.

Name: PROCESS_GROUP_REMOVED

Subtype Of: PROCESS_GROUP

Body Description: This body is a token set with just a groupname key/value.

```
groupname:cat
```

Extending Supervisor's XML-RPC API

Supervisor can be extended with new XML-RPC APIs. Several third-party plugins already exist that can be wired into your Supervisor configuration. You may additionally write your own. Extensible XML-RPC interfaces is an advanced feature, introduced in version 3.0. You needn't understand it unless you wish to use an existing third-party RPC interface plugin or if you wish to write your own RPC interface plugin.

Configuring XML-RPC Interface Factories

An additional RPC interface is configured into a supervisor installation by adding a `[rpcinterface:x]` section in the Supervisor configuration file.

In the sample config file, there is a section which is named `[rpcinterface:supervisor]`. By default it looks like this:

```
[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface
```

This section *must* remain in the configuration for the standard setup of supervisor to work properly. If you don't want supervisor to do anything it doesn't already do out of the box, this is all you need to know about this type of section.

However, if you wish to add additional XML-RPC interface namespaces to a configuration of supervisor, you may add additional `[rpcinterface:foo]` sections, where "foo" represents the namespace of the interface (from the web root), and the value named by `supervisor.rpcinterface_factory` is a factory callable written in Python which should have a function signature that accepts a single positional argument `supervisord` and as many keyword arguments as required to perform configuration. Any key/value pairs defined within the `rpcinterface:foo` section will be passed as keyword arguments to the factory. Here's an example of a factory function, created in the package `my.package`.

```
def make_another_rpcinterface(supervisord, **config):
    retries = int(config.get('retries', 0))
    another_rpc_interface = AnotherRPCInterface(supervisord, retries)
    return another_rpc_interface
```

And a section in the config file meant to configure it.

```
[rpcinterface:another]
supervisor.rpcinterface_factory = my.package:make_another_rpcinterface
retries = 1
```

Upgrading Supervisor 2 to 3

The following is true when upgrading an installation from Supervisor 2.X to Supervisor 3.X:

1. In `[program:x]` sections, the keys `logfile`, `logfile_backups`, `logfile_maxbytes`, `log_stderr` and `log_stdout` are no longer valid. Supervisor2 logged both stderr and stdout to a single log file. Supervisor 3 logs stderr and stdout to separate log files. You'll need to rename `logfile` to `stdout_logfile`, `logfile_backups` to `stdout_logfile_backups`, and `logfile_maxbytes` to `stdout_logfile_maxbytes` at the very least to preserve your configuration. If you created program sections where `log_stderr` was true, to preserve the behavior of sending stderr output to the stdout log, use the `redirect_stderr` boolean in the program section instead.

2. The supervisor configuration file must include the following section verbatim for the XML-RPC interface (and thus the web interface and **`supervisorctl`**) to work properly:

   ```
   [rpcinterface:supervisor]
   supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface
   ```

3. The semantics of the `autorestart` parameter within `[program:x]` sections has changed. This parameter used to accept only `true` or `false`. It now accepts an additional value, `unexpected`, which indicates that the process should restart from the `EXITED` state only if its exit code does not match any of those represented by the `exitcode` parameter in the process' configuration (implying a process crash). In addition, the default for `autorestart` is now `unexpected` (it used to be `true`, which meant restart unconditionally).

4. We now allow **`supervisord`** to listen on both a UNIX domain socket and an inet socket instead of making listening on one mutually exclusive with listening on the other. As a result, the options `http_port`, `http_username`, `http_password`, `sockchmod` and `sockchown` are no longer part of the `[supervisord]` section configuration. These have been supplanted by two other sections: `[unix_http_server]` and `[inet_http_server]`. You'll need to insert one or the other (depending on whether you want to listen on a UNIX domain socket or a TCP socket respectively) or both into your `supervisord.conf` file. These sections have their own options (where applicable) for `port`, `username`, `password`, `chmod`, and `chown`.

5. All supervisord command-line options related to `http_port`, `http_username`, `http_password`, `sockchmod` and `sockchown` have been removed (see above point for rationale).

6. The option that used to be `sockchown` within the `[supervisord]` section (and is now named `chown` within the `[unix_http_server]` section) used to accept a dot-separated (`user.group`) value. The separator now must be a colon, e.g. `user:group`. Unices allow for dots in usernames, so this change is a bugfix.

Frequently Asked Questions

Q My program never starts and supervisor doesn't indicate any error?

A Make sure the `x` bit is set on the executable file you're using in the `command=` line of your program section.

Q I am a software author and I want my program to behave differently when it's running under **`supervisord`**. How can I tell if my program is running under **`supervisord`**?

A Supervisor and its subprocesses share an environment variable `SUPERVISOR_ENABLED`. When your program is run under **`supervisord`**, it can check for the presence of this environment variable to determine whether it is running as a **`supervisord`** subprocess.

Q My command works fine when I invoke it by hand from a shell prompt, but when I use the same command line in a supervisor program `command=` section, the program fails mysteriously. Why?

A This may be due to your process' dependence on environment variable settings. See *Subprocess Environment*.

Q How can I make Supervisor restart a process that's using "too much" memory automatically?

A The *superlance* package contains a console script that can be used as a Supervisor event listener named `memmon` which helps with this task. It works on Linux and Mac OS X.

Resources and Development

Mailing Lists

Supervisor has a mailing list for users. You may subscribe to the Supervisor-users mailing list.

Supervisor has a mailing list for checkins too. You may subscribe to the Supervisor-checkins mailing list.

Bug Tracker

Supervisor has a bugtracker where you may report any bugs or other errors you find. Please report bugs to the GitHub issues page.

Version Control Repository

You can also view the Supervisor version control repository.

Contributing

Supervisor development is discussed on the mailing list. We'll review contributions from the community in both pull requests on GitHub (preferred) and patches sent to the list.

Author Information

The following people are responsible for creating Supervisor.

Original Author

- Chris McDonough is the original author of Supervisor.

Contributors

Contributors are tracked on the GitHub contributions page. The two lists below are included for historical reasons.

This first list recognizes significant contributions that were made before the repository moved to GitHub.

- Anders Quist: Anders contributed the patch that was the basis for Supervisor's ability to reload parts of its configuration without restarting.
- Derek DeVries: Derek did the web design of Supervisor's internal web interface and website logos.
- Guido van Rossum: Guido authored `zdrun` and `zdctl`, the programs from Zope that were the original basis for Supervisor. He also created Python, the programming language that Supervisor is written in.
- Jason Kirtland: Jason fixed Supervisor to run on Python 2.6 by contributing a patched version of Medusa (a Supervisor dependency) that we now bundle.
- Roger Hoover: Roger added support for spawning FastCGI programs. He has also been one of the most active mailing list users, providing his testing and feedback.
- Siddhant Goel: Siddhant worked on **`supervisorctl`** as our Google Summer of Code student for 2008. He implemented the `fg` command and also added tab completion.

This second list records contributors who signed a legal agreement. The legal agreement was introduced in January 2014 but later withdrawn in March 2014. This list is being preserved in case it is useful later (e.g. if at some point there was a desire to donate the project to a foundation that required such agreements).

- Chris McDonough, 2006-06-26
- Siddhant Goel, 2008-06-15
- Chris Rossi, 2010-02-02
- Roger Hoover, 2010-08-17
- Benoit Sigoure, 2011-06-21
- John Szakmeister, 2011-09-06
- Gunnlaugur Þór Briem, 2011-11-26
- Jens Rantil, 2011-11-27
- Michael Blume, 2012-01-09
- Philip Zeyliger, 2012-02-21
- Marcelo Vanzin, 2012-05-03
- Martijn Pieters, 2012-06-04
- Marcin Kuźmiński, 2012-06-21
- Jean Jordaan, 2012-06-28
- Perttu Ranta-aho, 2012-09-27
- Chris Streeter, 2013-03-23
- Caio Ariede, 2013-03-25
- David Birdsong, 2013-04-11

- Lukas Rist, 2013-04-18
- Honza Pokorny, 2013-07-23
- Thúlio Costa, 2013-10-31
- Gary M. Josack, 2013-11-12
- Márk Sági-Kazár, 2013-12-16

Glossary

daemontools A process control system by D.J. Bernstein.

launchd A process control system used by Apple as process 1 under Mac OS X.

runit A process control system.

Superlance A package which provides various event listener implementations that plug into Supervisor which can help monitor process memory usage and crash status: http://pypi.python.org/pypi/superlance.

umask Abbreviation of *user mask*: sets the file mode creation mask of the current process. See http://en.wikipedia.org/wiki/Umask.

CHAPTER 2

API Documentation

XML-RPC API Documentation

To use the XML-RPC interface, first make sure you have configured the interface factory properly by setting the default factory. See *Configuring XML-RPC Interface Factories*.

Then you can connect to supervisor's HTTP port with any XML-RPC client library and run commands against it. An example of doing this using Python's `xmlrpclib` client library is as follows.

```
import xmlrpclib
server = xmlrpclib.Server('http://localhost:9001/RPC2')
```

You may call methods against **supervisord** and its subprocesses by using the `supervisor` namespace. An example is provided below.

```
server.supervisor.getState()
```

You can get a list of methods supported by the **supervisord** XML-RPC interface by using the XML-RPC `system.listMethods` API:

```
server.system.listMethods()
```

You can see help on a method by using the `system.methodHelp` API against the method:

```
server.system.methodHelp('supervisor.shutdown')
```

The **supervisord** XML-RPC interface also supports the XML-RPC multicall API.

You can extend **supervisord** functionality with new XML-RPC API methods by adding new top-level RPC interfaces as necessary. See *Configuring XML-RPC Interface Factories*.

Note: Any XML-RPC method call may result in a fault response. This includes errors caused by the client such as bad arguments, and any errors that make **supervisord** unable to fulfill the request. Many XML-RPC client

programs will raise an exception when a fault response is encountered.

Status and Control

class supervisor.rpcinterface.**SupervisorNamespaceRPCInterface**(*supervisord*)

getAPIVersion()
Return the version of the RPC API used by supervisord

@return string version version id

This API is versioned separately from Supervisor itself. The API version returned by `getAPIVersion` only changes when the API changes. Its purpose is to help the client identify with which version of the Supervisor API it is communicating.

When writing software that communicates with this API, it is highly recommended that you first test the API version for compatibility before making method calls.

Note: The `getAPIVersion` method replaces `getVersion` found in Supervisor versions prior to 3.0a1. It is aliased for compatibility but getVersion() is deprecated and support will be dropped from Supervisor in a future version.

---

getSupervisorVersion()
Return the version of the supervisor package in use by supervisord

@return string version version id

getIdentification()
Return identifying string of supervisord

@return string identifier identifying string

This method allows the client to identify with which Supervisor instance it is communicating in the case of environments where multiple Supervisors may be running.

The identification is a string that must be set in Supervisor's configuration file. This method simply returns that value back to the client.

getState()
Return current state of supervisord as a struct

@return struct A struct with keys int statecode, string statename

This is an internal value maintained by Supervisor that determines what Supervisor believes to be its current operational state.

Some method calls can alter the current state of the Supervisor. For example, calling the method supervisor.shutdown() while the station is in the RUNNING state places the Supervisor in the SHUTDOWN state while it is shutting down.

The supervisor.getState() method provides a means for the client to check Supervisor's state, both for informational purposes and to ensure that the methods it intends to call will be permitted.

The return value is a struct:

```
{'statecode': 1,
 'statename': 'RUNNING'}
```

The possible return values are:

statecode	statename	Description
2	FATAL	Supervisor has experienced a serious error.
1	RUNNING	Supervisor is working normally.
0	RESTARTING	Supervisor is in the process of restarting.
-1	SHUTDOWN	Supervisor is in the process of shutting down.

The `FATAL` state reports unrecoverable errors, such as internal errors inside Supervisor or system runaway conditions. Once set to `FATAL`, the Supervisor can never return to any other state without being restarted.

In the `FATAL` state, all future methods except supervisor.shutdown() and supervisor.restart() will automatically fail without being called and the fault `FATAL_STATE` will be raised.

In the `SHUTDOWN` or `RESTARTING` states, all method calls are ignored and their possible return values are undefined.

getPID()
Return the PID of supervisord

@return int PID

readLog(*offset*, *length*)
Read length bytes from the main log starting at offset

@param int offset offset to start reading from. @param int length number of bytes to read from the log. @return string result Bytes of log

It can either return the entire log, a number of characters from the tail of the log, or a slice of the log specified by the offset and length parameters:

Offset	Length	Behavior of `readProcessLog`
Negative	Not Zero	Bad arguments. This will raise the fault `BAD_ARGUMENTS`.
Negative	Zero	This will return the tail of the log, or offset number of characters from the end of the log. For example, if `offset` = -4 and `length` = 0, then the last four characters will be returned from the end of the log.
Zero or Positive	Negative	Bad arguments. This will raise the fault `BAD_ARGUMENTS`.
Zero or Positive	Zero	All characters will be returned from the `offset` specified.
Zero or Positive	Positive	A number of characters length will be returned from the `offset`.

If the log is empty and the entire log is requested, an empty string is returned.

If either offset or length is out of range, the fault `BAD_ARGUMENTS` will be returned.

If the log cannot be read, this method will raise either the `NO_FILE` error if the file does not exist or the `FAILED` error if any other problem was encountered.

Note: The readLog() method replaces readMainLog() found in Supervisor versions prior to 2.1. It is aliased for compatibility but readMainLog() is deprecated and support will be dropped from Supervisor in a future version.

clearLog()
Clear the main log.

@return boolean result always returns True unless error

If the log cannot be cleared because the log file does not exist, the fault `NO_FILE` will be raised. If the log cannot be cleared for any other reason, the fault `FAILED` will be raised.

shutdown()
Shut down the supervisor process

@return boolean result always returns True unless error

This method shuts down the Supervisor daemon. If any processes are running, they are automatically killed without warning.

Unlike most other methods, if Supervisor is in the `FATAL` state, this method will still function.

restart()
Restart the supervisor process

@return boolean result always return True unless error

This method soft restarts the Supervisor daemon. If any processes are running, they are automatically killed without warning. Note that the actual UNIX process for Supervisor cannot restart; only Supervisor's main program loop. This has the effect of resetting the internal states of Supervisor.

Unlike most other methods, if Supervisor is in the `FATAL` state, this method will still function.

Process Control

class supervisor.rpcinterface.**SupervisorNamespaceRPCInterface**(*supervisord*)

getProcessInfo(*name*)
Get info about a process named name

@param string name The name of the process (or 'group:name') @return struct result A structure containing data about the process

The return value is a struct:

```
{'name':           'process name',
 'group':          'group name',
 'description':    'pid 18806, uptime 0:03:12'
 'start':          1200361776,
 'stop':           0,
 'now':            1200361812,
 'state':          1,
 'statename':      'RUNNING',
 'spawnerr':       '',
 'exitstatus':     0,
 'logfile':        '/path/to/stdout-log', # deprecated, b/c only
 'stdout_logfile': '/path/to/stdout-log',
```

```
 'stderr_logfile': '/path/to/stderr-log',
 'pid':            1}
```

name
Name of the process

group
Name of the process' group

description
If process state is running description's value is process_id and uptime. Example "pid 18806, uptime 0:03:12 ". If process state is stopped description's value is stop time. Example:"Jun 5 03:16 PM ".

start
UNIX timestamp of when the process was started

stop
UNIX timestamp of when the process last ended, or 0 if the process has never been stopped.

now
UNIX timestamp of the current time, which can be used to calculate process up-time.

state
State code, see *Process States*.

statename
String description of *state*, see *Process States*.

logfile
Deprecated alias for `stdout_logfile`. This is provided only for compatibility with clients written for Supervisor 2.x and may be removed in the future. Use `stdout_logfile` instead.

stdout_logfile
Absolute path and filename to the STDOUT logfile

stderr_logfile
Absolute path and filename to the STDOUT logfile

spawnerr
Description of error that occurred during spawn, or empty string if none.

exitstatus
Exit status (errorlevel) of process, or 0 if the process is still running.

pid
UNIX process ID (PID) of the process, or 0 if the process is not running.

getAllProcessInfo()
Get info about all processes

@return array result An array of process status results

Each element contains a struct, and this struct contains the exact same elements as the struct returned by `getProcessInfo`. If the process table is empty, an empty array is returned.

startProcess(*name*, *wait=True*)
Start a process

@param string name Process name (or `group:name`, or `group:*`) @param boolean wait Wait for process to be fully started @return boolean result Always true unless error

startAllProcesses(*wait=True*)
Start all processes listed in the configuration file

@param boolean wait Wait for each process to be fully started @return array result An array of process status info structs

startProcessGroup(*name*, *wait=True*)
Start all processes in the group named 'name'

@param string name The group name @param boolean wait Wait for each process to be fully started @return array result An array of process status info structs

stopProcess(*name*, *wait=True*)
Stop a process named by name

@param string name The name of the process to stop (or 'group:name') @param boolean wait Wait for the process to be fully stopped @return boolean result Always return True unless error

stopProcessGroup(*name*, *wait=True*)
Stop all processes in the process group named 'name'

@param string name The group name @param boolean wait Wait for each process to be fully stopped @return array result An array of process status info structs

stopAllProcesses(*wait=True*)
Stop all processes in the process list

@param boolean wait Wait for each process to be fully stopped @return array result An array of process status info structs

signalProcess(*name*, *signal*)
Send an arbitrary UNIX signal to the process named by name

@param string name Name of the process to signal (or 'group:name') @param string signal Signal to send, as name ('HUP') or number ('1') @return boolean

signalProcessGroup(*name*, *signal*)
Send a signal to all processes in the group named 'name'

@param string name The group name @param string signal Signal to send, as name ('HUP') or number ('1') @return array

signalAllProcesses(*signal*)
Send a signal to all processes in the process list

@param string signal Signal to send, as name ('HUP') or number ('1') @return array An array of process status info structs

sendProcessStdin(*name*, *chars*)
Send a string of chars to the stdin of the process name. If non-7-bit data is sent (unicode), it is encoded to utf-8 before being sent to the process' stdin. If chars is not a string or is not unicode, raise INCORRECT_PARAMETERS. If the process is not running, raise NOT_RUNNING. If the process' stdin cannot accept input (e.g. it was closed by the child process), raise NO_FILE.

@param string name The process name to send to (or 'group:name') @param string chars The character data to send to the process @return boolean result Always return True unless error

sendRemoteCommEvent(*type*, *data*)
Send an event that will be received by event listener subprocesses subscribing to the RemoteCommunicationEvent.

@param string type String for the "type" key in the event header @param string data Data for the event body @return boolean Always return True unless error

reloadConfig()
Reload the configuration.

The result contains three arrays containing names of process groups:
•*added* gives the process groups that have been added
•*changed* gives the process groups whose contents have changed
•*removed* gives the process groups that are no longer in the configuration
@return array result [[added, changed, removed]]

addProcessGroup(*name*)
Update the config for a running process from config file.

@param string name name of process group to add @return boolean result true if successful

removeProcessGroup(*name*)
Remove a stopped process from the active configuration.

@param string name name of process group to remove @return boolean result Indicates whether the removal was successful

Process Logging

class supervisor.rpcinterface.**SupervisorNamespaceRPCInterface**(*supervisord*)

readProcessStdoutLog(*name*, *offset*, *length*)
Read length bytes from name's stdout log starting at offset

@param string name the name of the process (or 'group:name') @param int offset offset to start reading from. @param int length number of bytes to read from the log. @return string result Bytes of log

readProcessStderrLog(*name*, *offset*, *length*)
Read length bytes from name's stderr log starting at offset

@param string name the name of the process (or 'group:name') @param int offset offset to start reading from. @param int length number of bytes to read from the log. @return string result Bytes of log

tailProcessStdoutLog(*name*, *offset*, *length*)
Provides a more efficient way to tail the (stdout) log than readProcessStdoutLog(). Use readProcessStdoutLog() to read chunks and tailProcessStdoutLog() to tail.

Requests (length) bytes from the (name)'s log, starting at (offset). If the total log size is greater than (offset + length), the overflow flag is set and the (offset) is automatically increased to position the buffer at the end of the log. If less than (length) bytes are available, the maximum number of available bytes will be returned. (offset) returned is always the last offset in the log +1.

@param string name the name of the process (or 'group:name') @param int offset offset to start reading from @param int length maximum number of bytes to return @return array result [string bytes, int offset, bool overflow]

tailProcessStderrLog(*name*, *offset*, *length*)
Provides a more efficient way to tail the (stderr) log than readProcessStderrLog(). Use readProcessStderrLog() to read chunks and tailProcessStderrLog() to tail.

Requests (length) bytes from the (name)'s log, starting at (offset). If the total log size is greater than (offset + length), the overflow flag is set and the (offset) is automatically increased to position the buffer at the end of the log. If less than (length) bytes are available, the maximum

number of available bytes will be returned. (offset) returned is always the last offset in the log +1.

@param string name the name of the process (or 'group:name') @param int offset offset to start reading from @param int length maximum number of bytes to return @return array result [string bytes, int offset, bool overflow]

clearProcessLogs(*name*)
Clear the stdout and stderr logs for the named process and reopen them.

@param string name The name of the process (or 'group:name') @return boolean result Always True unless error

clearAllProcessLogs()
Clear all process log files

@return array result An array of process status info structs

System Methods

class supervisor.xmlrpc.**SystemNamespaceRPCInterface**(*namespaces*)

listMethods()
Return an array listing the available method names

@return array result An array of method names available (strings).

methodHelp(*name*)
Return a string showing the method's documentation

@param string name The name of the method. @return string result The documentation for the method name.

methodSignature(*name*)
Return an array describing the method signature in the form [rtype, ptype, ptype...] where rtype is the return data type of the method, and ptypes are the parameter data types that the method accepts in method argument order.

@param string name The name of the method. @return array result The result.

multicall(*calls*)
Process an array of calls, and return an array of results. Calls should be structs of the form {'methodName': string, 'params': array}. Each result will either be a single-item array containing the result value, or a struct of the form {'faultCode': int, 'faultString': string}. This is useful when you need to make lots of small calls without lots of round trips.

@param array calls An array of call requests @return array result An array of results

CHAPTER 3

Plugins

Third Party Applications and Libraries

There are a number of third party applications that can be useful together with Supervisor. This list aims to summarize them and make them easier to find.

See README.rst for information on how to contribute to this list. Obviously, you can always also send an e-mail to the Supervisor mailing list to inform about missing plugins or libraries for/using Supervisor.

Dashboards and Tools for Multiple Supervisor Instances

These are tools that can monitor or control a number of Supervisor instances running on different servers.

cesi Web-based dashboard written in Python.

Django-Dashvisor Web-based dashboard written in Python. Requires Django 1.3 or 1.4.

Nodervisor Web-based dashboard written in Node.js.

Supervisord-Monitor Web-based dashboard written in PHP.

SupervisorUI Another Web-based dashboard written in PHP.

supervisorclusterctl Command line tool for controlling multiple Supervisor instances using Ansible.

suponoff Web-based dashboard written in Python 3. Requires Django 1.7 or later.

Supvisors Designed for distributed applications, written in Python 2.7. Includes an extended XML-RPC API and a Web-based dashboard.

Third Party Plugins and Libraries for Supervisor

These are plugins and libraries that add new functionality to Supervisor. These also includes various event listeners.

superlance Provides set of common eventlisteners that can be used to monitor and, for example, restart when it uses too much memory etc.

mr.rubber An event listener that makes it possible to scale the number of processes to the number of cores on the supervisor host.

supervisor-wildcards Implements start/stop/restart commands with wildcard support for Supervisor. These commands run in parallel and can be much faster than the built-in start/stop/restart commands.

mr.laforge Lets you easily make sure that `supervisord` and specific processes controlled by it are running from within shell and Python scripts. Also adds a `kill` command to supervisor that makes it possible to send arbitrary signals to child processes.

supervisor_cache An extension for Supervisor that provides the ability to cache arbitrary data directly inside a Supervisor instance as key/value pairs. Also serves as a reference for how to write Supervisor extensions.

supervisor_twiddler An RPC extension for Supervisor that allows Supervisor's configuration and state to be manipulated in ways that are not normally possible at runtime.

supervisor-stdout An event listener that sends process output to supervisord's stdout.

supervisor-serialrestart Adds a `serialrestart` command to `supervisorctl` that restarts processes one after another rather than all at once.

supervisor-quick Adds `quickstart`, `quickstop`, and `quickrestart` commands to `supervisorctl` that can be faster than the built-in commands. It works by using the non-blocking mode of the XML-RPC methods and then polling `supervisord`. The built-in commands use the blocking mode, which can be slower due to `supervisord` implementation details.

supervisor-logging An event listener that sends process log events to an external Syslog instance (e.g. Logstash).

supervisor-logstash-notifier An event listener plugin to stream state events to a Logstash instance.

supervisor_cgroups An event listener that enables tying Supervisor processes to a cgroup hierarchy. It is intended to be used as a replacement for cgrules.conf.

supervisor_checks Framework to build health checks for Supervisor-based services. Health check applications are supposed to run as event listeners in Supervisor environment. On check failure Supervisor will attempt to restart monitored process.

Libraries that integrate Third Party Applications with Supervisor

These are libraries and plugins that makes it easier to use Supervisor with third party applications:

django-supervisor Easy integration between djangocl and supervisord.

collective.recipe.supervisor A buildout recipe to install supervisor.

puppet-module-supervisor Puppet module for configuring the supervisor daemon tool.

puppet-supervisord Puppet module to manage the supervisord process control system.

ngx_supervisord An nginx module providing API to communicate with supervisord and manage (start/stop) backends on-demand.

Supervisord-Nagios-Plugin A Nagios/Icinga plugin written in Python to monitor individual supervisord processes.

nagios-supervisord-processes A Nagios/Icinga plugin written in PHP to monitor individual supervisord processes.

supervisord-nagios A plugin for supervisorctl to allow one to perform nagios-style checks against supervisord-managed processes.

php-supervisor-event PHP classes for interacting with Supervisor event notifications.

PHP5 Supervisor wrapper PHP 5 library to manage Supervisor instances as object.

Symfony2 SupervisorBundle Provide full integration of Supervisor multiple servers management into Symfony2 project.

sd-supervisord Server Density plugin for supervisor.

node-supervisord Node.js client for Supervisor's XML-RPC interface.

node-supervisord-eventlistener Node.js implementation of an event listener for Supervisor.

ruby-supervisor Ruby client library for Supervisor's XML-RPC interface.

Sulphite Sends supervisord events to Graphite.

supervisord.tmbundle TextMate bundle for supervisord.conf.

capistrano-supervisord Capistrano recipe to deploy supervisord based services.

capistrano-supervisor Another package to control supervisord from Capistrano.

chef-supervisor Chef cookbook install and configure supervisord.

SupervisorPHP Complete Supervisor suite in PHP: Client using XML-RPC interface, event listener and configuration builder implementation, console application and monitor UI.

Supervisord-Client Perl client for the supervisord XML-RPC interface.

supervisord4j Java client for Supervisor's XML-RPC interface.

Supermann Supermann monitors processes running under Supervisor and sends metrics to Riemann.

gulp-supervisor Run Supervisor as a Gulp task.

Yeebase.Supervisor Control and monitor Supervisor from a TYPO3 Flow application.

dokku-supervisord Dokku plugin that injects `supervisord` to run applications.

dokku-logging-supervisord Dokku plugin that injects `supervisord` to run applications. It also redirects `stdout` and `stderr` from processes to log files (rather than the Docker default per-container JSON files).

superslacker Send Supervisor event notifications to Slack.

supervisor-alert Send event notifications over Telegram or to an arbitrary command.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

S

T

U