BuildBot

This is the BuildBot manual.

Table of Contents

1 Introduction

The BuildBot is a system to automate the compile/test cycle required by most software projects to validate code changes. By automatically rebuilding and testing the tree each time something has changed, build problems are pinpointed quickly, before other developers are inconvenienced by the failure. The guilty developer can be identified and harassed without human intervention. By running the builds on a variety of platforms, developers who do not have the facilities to test their changes everywhere before checkin will at least know shortly afterwards whether they have broken the build or not. Warning counts, lint checks, image size, compile time, and other build parameters can be tracked over time, are more visible, and are therefore easier to improve.

The overall goal is to reduce tree breakage and provide a platform to run tests or code-quality checks that are too annoying or pedantic for any human to waste their time with. Developers get immediate (and potentially public) feedback about their changes, encouraging them to be more careful about testing before checkin.

Features:

- run builds on a variety of slave platforms
- arbitrary build process: handles projects using C, Python, whatever
- minimal host requirements: python and Twisted
- slaves can be behind a firewall if they can still do checkout
- status delivery through web page, email, IRC, other protocols
- track builds in progress, provide estimated completion time
- flexible configuration by subclassing generic build process classes
- debug tools to force a new build, submit fake Changes, query slave status
- released under the GPL

1.1 History and Philosophy

The Buildbot was inspired by a similar project built for a development team writing a cross-platform embedded system. The various components of the project were supposed to compile and run on several flavors of unix (linux, solaris, BSD), but individual developers had their own preferences and tended to stick to a single platform. From time to time, incompatibilities would sneak in (some unix platforms want to use `string.h`, some prefer `strings.h`), and then the tree would compile for some developers but not others. The buildbot was written to automate the human process of walking into the office, updating a tree, compiling (and discovering the breakage), finding the developer at fault, and complaining to them about the problem they had introduced. With multiple platforms it was difficult for developers to do the right thing (compile their potential change on all platforms); the buildbot offered a way to help.

Another problem was when programmers would change the behavior of a library without warning its users, or change internal aspects that other code was (unfortunately) depending upon. Adding unit tests to the codebase helps here: if an application's unit tests pass despite changes in the libraries it uses, you can have more confidence that the library changes haven't broken anything. Many developers complained that the unit tests were

inconvenient or took too long to run: having the buildbot run them reduces the developer's workload to a minimum.

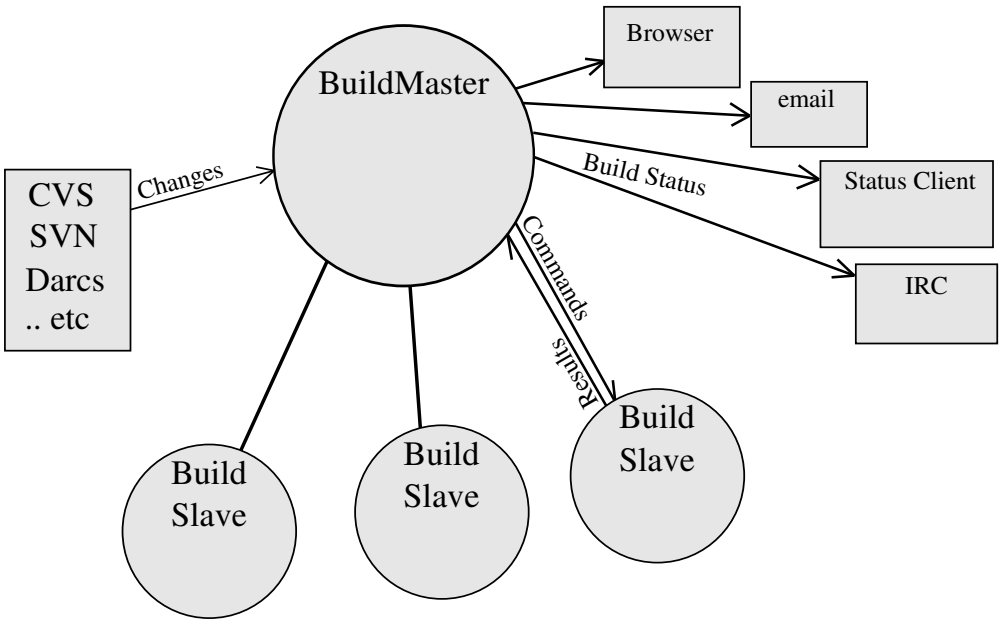
In general, having more visibility into the project is always good, and automation makes it easier for developers to do the right thing. When everyone can see the status of the project, developers are encouraged to keep the tree in good working order. Unit tests that aren't run on a regular basis tend to suffer from bitrot just like code does: exercising them on a regular basis helps to keep them functioning and useful.

The current version of the Buildbot is additionally targeted at distributed free-software projects, where resources and platforms are only available when provided by interested volunteers. The buildslaves are designed to require an absolute minimum of configuration, reducing the effort a potential volunteer needs to expend to be able to contribute a new test environment to the project. The goal is for anyone who wishes that a given project would run on their favorite platform should be able to offer that project a buildslave, running on that platform, where they can verify that their portability code works, and keeps working.

1.2 System Architecture

The Buildbot consists of a single `buildmaster` and one or more `buildslaves`, connected in a star topology. The buildmaster makes all decisions about what, when, and how to build. It sends commands to be run on the build slaves, which simply execute the commands and return the results. (certain steps involve more local decision making, where the overhead of sending a lot of commands back and forth would be inappropriate, but in general the buildmaster is responsible for everything).

The buildmaster is usually fed `Changes` by some sort of version control system (see Section 5.1 [Change Sources], page 50), which may cause builds to be run. As the builds are performed, various status messages are produced, which are then sent to any registered Status Targets (see Chapter 7 [Status Delivery], page 105).

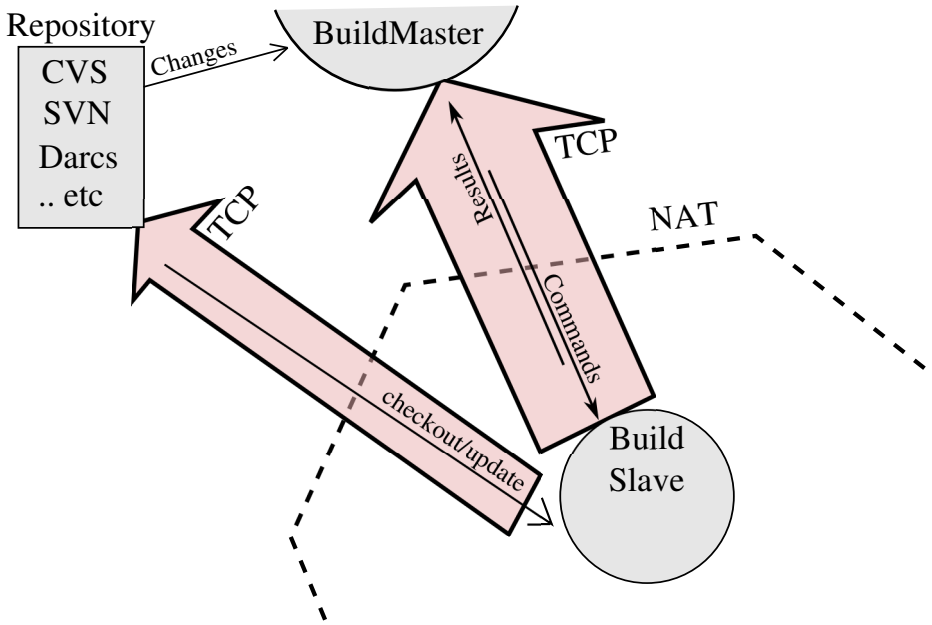


The buildmaster is configured and maintained by the "buildmaster admin", who is generally the project team member responsible for build process issues. Each buildslave is maintained by a "buildslave admin", who do not need to be quite as involved. Generally slaves are run by anyone who has an interest in seeing the project work well on their favorite platform.

1.2.1 BuildSlave Connections

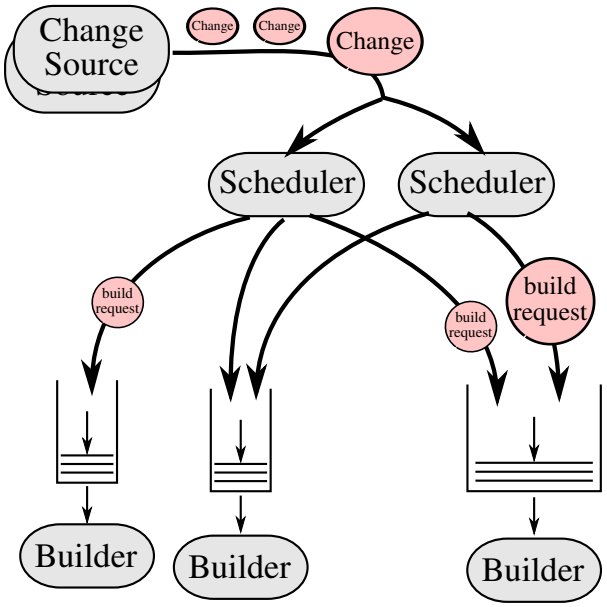
The buildslaves are typically run on a variety of separate machines, at least one per platform of interest. These machines connect to the buildmaster over a TCP connection to a publically-visible port. As a result, the buildslaves can live behind a NAT box or similar firewalls, as long as they can get to buildmaster. The TCP connections are initiated by the buildslave and accepted by the buildmaster, but commands and results travel both ways within this connection. The buildmaster is always in charge, so all commands travel exclusively from the buildmaster to the buildslave.

To perform builds, the buildslaves must typically obtain source code from a CVS/SVN/etc repository. Therefore they must also be able to reach the repository. The buildmaster provides instructions for performing builds, but does not provide the source code itself.



1.2.2 Buildmaster Architecture

The Buildmaster consists of several pieces:

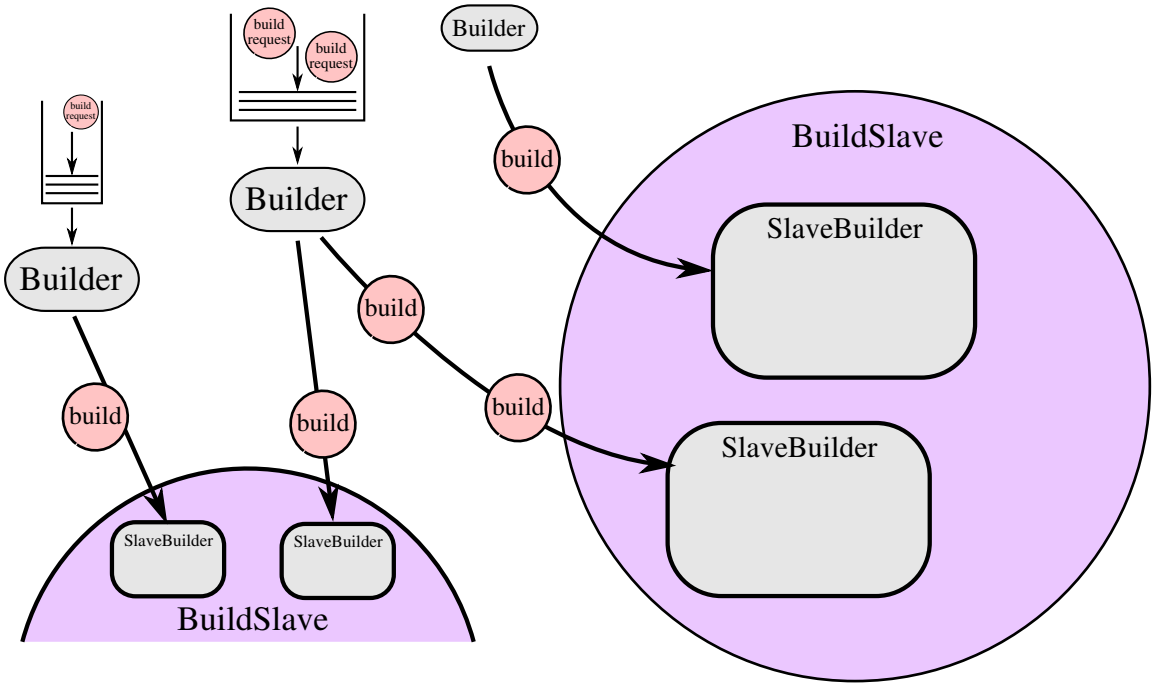


- Change Sources, which create a Change object each time something is modified in the VC repository. Most ChangeSources listen for messages from a hook script of some sort. Some sources actively poll the repository on a regular basis. All Changes are fed to the Schedulers.

- Schedulers, which decide when builds should be performed. They collect Changes into BuildRequests, which are then queued for delivery to Builders until a buildslave is available.

- Builders, which control exactly *how* each build is performed (with a series of BuildSteps, configured in a BuildFactory). Each Build is run on a single buildslave.

- Status plugins, which deliver information about the build results through protocols like HTTP, mail, and IRC.



Each Builder is configured with a list of BuildSlaves that it will use for its builds. These buildslaves are expected to behave identically: the only reason to use multiple BuildSlaves for a single Builder is to provide a measure of load-balancing.

Within a single BuildSlave, each Builder creates its own SlaveBuilder instance. These SlaveBuilders operate independently from each other. Each gets its own base directory to work in. It is quite common to have many Builders sharing the same buildslave. For example, there might be two buildslaves: one for i386, and a second for PowerPC. There may then be a pair of Builders that do a full compile/test run, one for each architecture, and a lone Builder that creates snapshot source tarballs if the full builders complete successfully. The full builders would each run on a single buildslave, whereas the tarball creation step might run on either buildslave (since the platform doesn't matter when creating source tarballs). In this case, the mapping would look like:

```
Builder(full-i386)  ->  BuildSlaves(slave-i386)
Builder(full-ppc)   ->  BuildSlaves(slave-ppc)
Builder(source-tarball) -> BuildSlaves(slave-i386, slave-ppc)
```

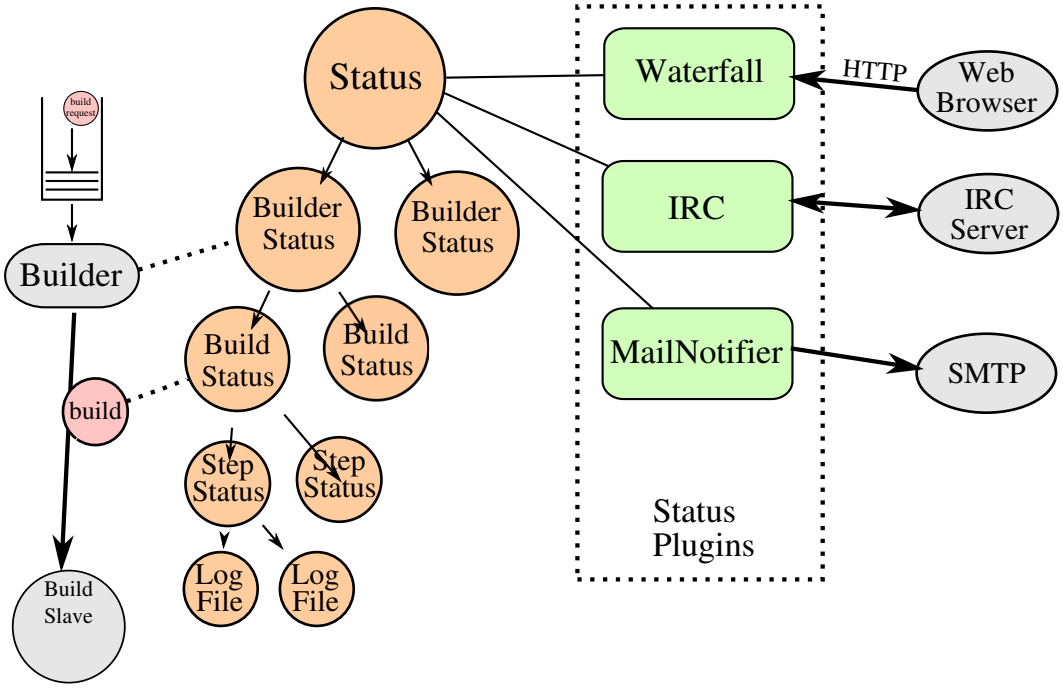
and each BuildSlave would have two SlaveBuilders inside it, one for a full builder, and a second for the source-tarball builder.

Once a SlaveBuilder is available, the Builder pulls one or more BuildRequests off its incoming queue. (It may pull more than one if it determines that it can merge the requests together; for example, there may be multiple requests to build the current HEAD revision). These requests are merged into a single Build instance, which includes the SourceStamp that describes what exact version of the source code should be used for the build. The Build is then randomly assigned to a free SlaveBuilder and the build begins.

The behaviour when BuildRequests are merged can be customized, see Section 4.6 [Merging BuildRequests], page 38.

1.2.3 Status Delivery Architecture

The buildmaster maintains a central Status object, to which various status plugins are connected. Through this Status object, a full hierarchy of build status objects can be obtained.



The configuration file controls which status plugins are active. Each status plugin gets a reference to the top-level Status object. From there they can request information on each Builder, Build, Step, and LogFile. This query-on-demand interface is used by the html.Waterfall plugin to create the main status page each time a web browser hits the main URL.

The status plugins can also subscribe to hear about new Builds as they occur: this is used by the MailNotifier to create new email messages for each recently-completed Build.

The Status object records the status of old builds on disk in the buildmaster's base directory. This allows it to return information about historical builds.

There are also status objects that correspond to Schedulers and BuildSlaves. These allow status plugins to report information about upcoming builds, and the online/offline status of each buildslave.

1.3 Control Flow

A day in the life of the buildbot:

- A developer commits some source code changes to the repository. A hook script or commit trigger of some sort sends information about this change to the buildmaster through one of its configured Change Sources. This notification might arrive via email, or over a network connection (either initiated by the buildmaster as it "subscribes" to changes, or by the commit trigger as it pushes Changes towards the buildmaster). The

Change contains information about who made the change, what files were modified, which revision contains the change, and any checkin comments.

- The buildmaster distributes this change to all of its configured Schedulers. Any "important" changes cause the "tree-stable-timer" to be started, and the Change is added to a list of those that will go into a new Build. When the timer expires, a Build is started on each of a set of configured Builders, all compiling/testing the same source code. Unless configured otherwise, all Builds run in parallel on the various buildslaves.
- The Build consists of a series of Steps. Each Step causes some number of commands to be invoked on the remote buildslave associated with that Builder. The first step is almost always to perform a checkout of the appropriate revision from the same VC system that produced the Change. The rest generally perform a compile and run unit tests. As each Step runs, the buildslave reports back command output and return status to the buildmaster.
- As the Build runs, status messages like "Build Started", "Step Started", "Build Finished", etc, are published to a collection of Status Targets. One of these targets is usually the HTML "Waterfall" display, which shows a chronological list of events, and summarizes the results of the most recent build at the top of each column. Developers can periodically check this page to see how their changes have fared. If they see red, they know that they've made a mistake and need to fix it. If they see green, they know that they've done their duty and don't need to worry about their change breaking anything.
- If a MailNotifier status target is active, the completion of a build will cause email to be sent to any developers whose Changes were incorporated into this Build. The MailNotifier can be configured to only send mail upon failing builds, or for builds which have just transitioned from passing to failing. Other status targets can provide similar real-time notification via different communication channels, like IRC.

2 Installation

2.1 Requirements

At a bare minimum, you'll need the following (for both the buildmaster and a buildslave):

- Python: http://www.python.org

  Buildbot requires python-2.3 or later, and is primarily developed against python-2.4. It is also tested against python-2.5 .

- Twisted: http://twistedmatrix.com

  Both the buildmaster and the buildslaves require Twisted-2.0.x or later. It has been tested against all releases of Twisted up to Twisted-2.5.0 (the most recent as of this writing). As always, the most recent version is recommended.

  Twisted is delivered as a collection of subpackages. You'll need at least "Twisted" (the core package), and you'll also want TwistedMail, TwistedWeb, and TwistedWords (for sending email, serving a web status page, and delivering build status via IRC, respectively). You might also want TwistedConch (for the encrypted Manhole debug port). Note that Twisted requires ZopeInterface to be installed as well.

Certain other packages may be useful on the system running the buildmaster:

- CVSToys: http://purl.net/net/CVSToys

  If your buildmaster uses FreshCVSSource to receive change notification from a cvstoys daemon, it will require CVSToys be installed (tested with CVSToys-1.0.10). If the it doesn't use that source (i.e. if you only use a mail-parsing change source, or the SVN notification script), you will not need CVSToys.

And of course, your project's build process will impose additional requirements on the buildslaves. These hosts must have all the tools necessary to compile and test your project's source code.

2.2 Installing the code

The Buildbot is installed using the standard python `distutils` module. After unpacking the tarball, the process is:

```
python setup.py build
python setup.py install
```

where the install step may need to be done as root. This will put the bulk of the code in somewhere like /usr/lib/python2.3/site-packages/buildbot . It will also install the `buildbot` command-line tool in /usr/bin/buildbot.

To test this, shift to a different directory (like /tmp), and run:

```
buildbot --version
```

If it shows you the versions of Buildbot and Twisted, the install went ok. If it says `no such command` or it gets an `ImportError` when it tries to load the libaries, then something went wrong. `pydoc buildbot` is another useful diagnostic tool.

Windows users will find these files in other places. You will need to make sure that python can find the libraries, and will probably find it convenient to have `buildbot` on your PATH.

If you wish, you can run the buildbot unit test suite like this:

```
PYTHONPATH=. trial buildbot.test
```

This should run up to 192 tests, depending upon what VC tools you have installed. On my desktop machine it takes about five minutes to complete. Nothing should fail, a few might be skipped. If any of the tests fail, you should stop and investigate the cause before continuing the installation process, as it will probably be easier to track down the bug early.

If you cannot or do not wish to install the buildbot into a site-wide location like '`/usr`' or '`/usr/local`', you can also install it into the account's home directory. Do the install command like this:

```
python setup.py install --home=~
```

That will populate '`~/lib/python`' and create '`~/bin/buildbot`'. Make sure this lib directory is on your `PYTHONPATH`.

2.3 Creating a buildmaster

As you learned earlier (see Section 1.2 [System Architecture], page 2), the buildmaster runs on a central host (usually one that is publically visible, so everybody can check on the status of the project), and controls all aspects of the buildbot system. Let us call this host `buildbot.example.org`.

You may wish to create a separate user account for the buildmaster, perhaps named `buildmaster`. This can help keep your personal configuration distinct from that of the buildmaster and is useful if you have to use a mail-based notification system (see Section 5.1 [Change Sources], page 50). However, the Buildbot will work just fine with your regular user account.

You need to choose a directory for the buildmaster, called the `basedir`. This directory will be owned by the buildmaster, which will use configuration files therein, and create status files as it runs. '`~/Buildbot`' is a likely value. If you run multiple buildmasters in the same account, or if you run both masters and slaves, you may want a more distinctive name like '`~/Buildbot/master/gnomovision`' or '`~/Buildmasters/fooproject`'. If you are using a separate user account, this might just be '`~buildmaster/masters/fooproject`'.

Once you've picked a directory, use the `buildbot create-master` command to create the directory and populate it with startup files:

```
buildbot create-master basedir
```

You will need to create a configuration file (see Chapter 4 [Configuration], page 29) before starting the buildmaster. Most of the rest of this manual is dedicated to explaining how to do this. A sample configuration file is placed in the working directory, named '`master.cfg.sample`', which can be copied to '`master.cfg`' and edited to suit your purposes.

(Internal details: This command creates a file named '`buildbot.tac`' that contains all the state necessary to create the buildmaster. Twisted has a tool called `twistd` which can use this .tac file to create and launch a buildmaster instance. twistd takes care of logging and daemonization (running the program in the background). '`/usr/bin/buildbot`' is a front end which runs twistd for you.)

In addition to '`buildbot.tac`', a small '`Makefile.sample`' is installed. This can be used as the basis for customized daemon startup, See Section 2.6 [Launching the daemons], page 13.

2.4 Upgrading an Existing Buildmaster

If you have just installed a new version of the Buildbot code, and you have buildmasters that were created using an older version, you'll need to upgrade these buildmasters before you can use them. The upgrade process adds and modifies files in the buildmaster's base directory to make it compatible with the new code.

```
buildbot upgrade-master basedir
```

This command will also scan your '`master.cfg`' file for incompatibilities (by loading it and printing any errors or deprecation warnings that occur). Each buildbot release tries to be compatible with configurations that worked cleanly (i.e. without deprecation warnings) on the previous release: any functions or classes that are to be removed will first be deprecated in a release, to give users a chance to start using their replacement.

The 0.7.6 release introduced the '`public_html/`' directory, which contains '`index.html`' and other files served by the `WebStatus` and `Waterfall` status displays. The `upgrade-master` command will create these files if they do not already exist. It will not modify existing copies, but it will write a new copy in e.g. '`index.html.new`' if the new version differs from the version that already exists.

The `upgrade-master` command is idempotent. It is safe to run it multiple times. After each upgrade of the buildbot code, you should use `upgrade-master` on all your buildmasters.

2.5 Creating a buildslave

Typically, you will be adding a buildslave to an existing buildmaster, to provide additional architecture coverage. The buildbot administrator will give you several pieces of information necessary to connect to the buildmaster. You should also be somewhat familiar with the project being tested, so you can troubleshoot build problems locally.

The buildbot exists to make sure that the project's stated "how to build it" process actually works. To this end, the buildslave should run in an environment just like that of your regular developers. Typically the project build process is documented somewhere ('`README`', '`INSTALL`', etc), in a document that should mention all library dependencies and contain a basic set of build instructions. This document will be useful as you configure the host and account in which the buildslave runs.

Here's a good checklist for setting up a buildslave:

1. Set up the account

   It is recommended (although not mandatory) to set up a separate user account for the buildslave. This account is frequently named `buildbot` or `buildslave`. This serves to isolate your personal working environment from that of the slave's, and helps to minimize the security threat posed by letting possibly-unknown contributors run arbitrary code on your system. The account should have a minimum of fancy init scripts.

2. Install the buildbot code

   Follow the instructions given earlier (see Section 2.2 [Installing the code], page 8). If you use a separate buildslave account, and you didn't install the buildbot code to a shared location, then you will need to install it with `--home=~` for each account that needs it.

3. Set up the host

   Make sure the host can actually reach the buildmaster. Usually the buildmaster is running a status webserver on the same machine, so simply point your web browser at it and see if you can get there. Install whatever additional packages or libraries the project's INSTALL document advises. (or not: if your buildslave is supposed to make sure that building without optional libraries still works, then don't install those libraries).

   Again, these libraries don't necessarily have to be installed to a site-wide shared location, but they must be available to your build process. Accomplishing this is usually very specific to the build process, so installing them to '`/usr`' or '`/usr/local`' is usually the best approach.

4. Test the build process

   Follow the instructions in the INSTALL document, in the buildslave's account. Perform a full CVS (or whatever) checkout, configure, make, run tests, etc. Confirm that the build works without manual fussing. If it doesn't work when you do it by hand, it will be unlikely to work when the buildbot attempts to do it in an automated fashion.

5. Choose a base directory

   This should be somewhere in the buildslave's account, typically named after the project which is being tested. The buildslave will not touch any file outside of this directory. Something like '`~/Buildbot`' or '`~/Buildslaves/fooproject`' is appropriate.

6. Get the buildmaster host/port, botname, and password

   When the buildbot admin configures the buildmaster to accept and use your buildslave, they will provide you with the following pieces of information:

   - your buildslave's name
   - the password assigned to your buildslave
   - the hostname and port number of the buildmaster, i.e. buildbot.example.org:8007

7. Create the buildslave

   Now run the 'buildbot' command as follows:

   `buildbot create-slave` *`BASEDIR MASTERHOST`*`:`*`PORT SLAVENAME PASSWORD`*

   This will create the base directory and a collection of files inside, including the '`buildbot.tac`' file that contains all the information you passed to the `buildbot` command.

8. Fill in the hostinfo files

   When it first connects, the buildslave will send a few files up to the buildmaster which describe the host that it is running on. These files are presented on the web status display so that developers have more information to reproduce any test failures that are witnessed by the buildbot. There are sample files in the '`info`' subdirectory of the

buildbot's base directory. You should edit these to correctly describe you and your host.

'`BASEDIR/info/admin`' should contain your name and email address. This is the "buildslave admin address", and will be visible from the build status page (so you may wish to munge it a bit if address-harvesting spambots are a concern).

'`BASEDIR/info/host`' should be filled with a brief description of the host: OS, version, memory size, CPU speed, versions of relevant libraries installed, and finally the version of the buildbot code which is running the buildslave.

The optional '`BASEDIR/info/access_uri`' can specify a URI which will connect a user to the machine. Many systems accept `ssh://hostname` URIs for this purpose.

If you run many buildslaves, you may want to create a single '`~buildslave/info`' file and share it among all the buildslaves with symlinks.

2.5.1 Buildslave Options

There are a handful of options you might want to use when creating the buildslave with the `buildbot create-slave <options> DIR <params>` command. You can type `buildbot create-slave --help` for a summary. To use these, just include them on the `buildbot create-slave` command line, like this:

```
buildbot create-slave --umask=022 ~/buildslave buildmaster.example.org:42012 myslav
```

`--usepty` This is a boolean flag that tells the buildslave whether to launch child processes in a PTY or with regular pipes (the default) when the master does not specify. This option is deprecated, as this particular parameter is better specified on the master.

`--umask` This is a string (generally an octal representation of an integer) which will cause the buildslave process' "umask" value to be set shortly after initialization. The "twistd" daemonization utility forces the umask to 077 at startup (which means that all files created by the buildslave or its child processes will be unreadable by any user other than the buildslave account). If you want build products to be readable by other accounts, you can add `--umask=022` to tell the buildslave to fix the umask after twistd clobbers it. If you want build products to be *writable* by other accounts too, use `--umask=000`, but this is likely to be a security problem.

`--keepalive`
This is a number that indicates how frequently "keepalive" messages should be sent from the buildslave to the buildmaster, expressed in seconds. The default (600) causes a message to be sent to the buildmaster at least once every 10 minutes. To set this to a lower value, use e.g. `--keepalive=120`.

If the buildslave is behind a NAT box or stateful firewall, these messages may help to keep the connection alive: some NAT boxes tend to forget about a connection if it has not been used in a while. When this happens, the buildmaster will think that the buildslave has disappeared, and builds will time out. Meanwhile the buildslave will not realize than anything is wrong.

`--maxdelay`
: This is a number that indicates the maximum amount of time the buildslave will wait between connection attempts, expressed in seconds. The default (300) causes the buildslave to wait at most 5 minutes before trying to connect to the buildmaster again.

`--log-size`
: This is the size in bytes when to rotate the Twisted log files.

`--log-count`
: This is the number of log rotations to keep around. You can either specify a number or `None` (the default) to keep all '`twistd.log`' files around.

2.6 Launching the daemons

Both the buildmaster and the buildslave run as daemon programs. To launch them, pass the working directory to the `buildbot` command:

```
buildbot start BASEDIR
```

The *BASEDIR* is option and can be omitted if the current directory contains the buildbot configuration (the '`buildbot.tac`' file).

```
buildbot start
```

This command will start the daemon and then return, so normally it will not produce any output. To verify that the programs are indeed running, look for a pair of files named '`twistd.log`' and '`twistd.pid`' that should be created in the working directory. '`twistd.pid`' contains the process ID of the newly-spawned daemon.

When the buildslave connects to the buildmaster, new directories will start appearing in its base directory. The buildmaster tells the slave to create a directory for each Builder which will be using that slave. All build operations are performed within these directories: CVS checkouts, compiles, and tests.

Once you get everything running, you will want to arrange for the buildbot daemons to be started at boot time. One way is to use `cron`, by putting them in a `@`reboot crontab entry[1]:

```
@reboot buildbot start BASEDIR
```

When you run `crontab` to set this up, remember to do it as the buildmaster or buildslave account! If you add this to your crontab when running as your regular account (or worse yet, root), then the daemon will run as the wrong user, quite possibly as one with more authority than you intended to provide.

It is important to remember that the environment provided to cron jobs and init scripts can be quite different that your normal runtime. There may be fewer environment variables specified, and the PATH may be shorter than usual. It is a good idea to test out this method of launching the buildslave by using a cron job with a time in the near future, with the same command, and then check '`twistd.log`' to make sure the slave actually started correctly. Common problems here are for '`/usr/local`' or '`~/bin`' to not be on your `PATH`, or for `PYTHONPATH` to not be set correctly. Sometimes `HOME` is messed up too.

[1] this `@`reboot syntax is understood by Vixie cron, which is the flavor usually provided with linux systems. Other unices may have a cron that doesn't understand `@`reboot

To modify the way the daemons are started (perhaps you want to set some environment variables first, or perform some cleanup each time), you can create a file named '`Makefile.buildbot`' in the base directory. When the '`buildbot`' front-end tool is told to `start` the daemon, and it sees this file (and '`/usr/bin/make`' exists), it will do `make -f Makefile.buildbot start` instead of its usual action (which involves running `twistd`). When the buildmaster or buildslave is installed, a '`Makefile.sample`' is created which implements the same behavior as the the '`buildbot`' tool uses, so if you want to customize the process, just copy '`Makefile.sample`' to '`Makefile.buildbot`' and edit it as necessary.

Some distributions may include conveniences to make starting buildbot at boot time easy. For instance, with the default buildbot package in Debian-based distributions, you may only need to modify `/etc/default/buildbot` (see also `/etc/init.d/buildbot`, which reads the configuration in `/etc/default/buildbot`).

2.7 Logfiles

While a buildbot daemon runs, it emits text to a logfile, named '`twistd.log`'. A command like `tail -f twistd.log` is useful to watch the command output as it runs.

The buildmaster will announce any errors with its configuration file in the logfile, so it is a good idea to look at the log at startup time to check for any problems. Most buildmaster activities will cause lines to be added to the log.

2.8 Shutdown

To stop a buildmaster or buildslave manually, use:

```
buildbot stop BASEDIR
```

This simply looks for the '`twistd.pid`' file and kills whatever process is identified within.

At system shutdown, all processes are sent a `SIGKILL`. The buildmaster and buildslave will respond to this by shutting down normally.

The buildmaster will respond to a `SIGHUP` by re-reading its config file. Of course, this only works on unix-like systems with signal support, and won't work on Windows. The following shortcut is available:

```
buildbot reconfig BASEDIR
```

When you update the Buildbot code to a new release, you will need to restart the buildmaster and/or buildslave before it can take advantage of the new code. You can do a `buildbot stop` *BASEDIR* and `buildbot start` *BASEDIR* in quick succession, or you can use the `restart` shortcut, which does both steps for you:

```
buildbot restart BASEDIR
```

There are certain configuration changes that are not handled cleanly by `buildbot reconfig`. If this occurs, `buildbot restart` is a more robust tool to fully switch over to the new configuration.

`buildbot restart` may also be used to start a stopped Buildbot instance. This behaviour is useful when writing scripts that stop, start and restart Buildbot.

A buildslave may also be gracefully shutdown from the see Section 7.1 [WebStatus], page 105 status plugin. This is useful to shutdown a buildslave without interrupting any current builds. The buildmaster will wait until the buildslave is finished all its current builds, and will then tell the buildslave to shutdown.

2.9 Maintenance

It is a good idea to check the buildmaster's status page every once in a while, to see if your buildslave is still online. Eventually the buildbot will probably be enhanced to send you email (via the '`info/admin`' email address) when the slave has been offline for more than a few hours.

If you find you can no longer provide a buildslave to the project, please let the project admins know, so they can put out a call for a replacement.

The Buildbot records status and logs output continually, each time a build is performed. The status tends to be small, but the build logs can become quite large. Each build and log are recorded in a separate file, arranged hierarchically under the buildmaster's base directory. To prevent these files from growing without bound, you should periodically delete old build logs. A simple cron job to delete anything older than, say, two weeks should do the job. The only trick is to leave the '`buildbot.tac`' and other support files alone, for which find's `-mindepth` argument helps skip everything in the top directory. You can use something like the following:

```
@weekly cd BASEDIR && find . -mindepth 2 i-path './public_html/*' -prune -o -type f
@weekly cd BASEDIR && find twistd.log* -mtime +14 -exec rm {} \;
```

2.10 Troubleshooting

Here are a few hints on diagnosing common problems.

2.10.1 Starting the buildslave

Cron jobs are typically run with a minimal shell ('`/bin/sh`', not '`/bin/bash`'), and tilde expansion is not always performed in such commands. You may want to use explicit paths, because the `PATH` is usually quite short and doesn't include anything set by your shell's startup scripts ('`.profile`', '`.bashrc`', etc). If you've installed buildbot (or other python libraries) to an unusual location, you may need to add a `PYTHONPATH` specification (note that python will do tilde-expansion on `PYTHONPATH` elements by itself). Sometimes it is safer to fully-specify everything:

```
@reboot PYTHONPATH=~/lib/python /usr/local/bin/buildbot start /usr/home/buildbot/ba
```

Take the time to get the `@reboot` job set up. Otherwise, things will work fine for a while, but the first power outage or system reboot you have will stop the buildslave with nothing but the cries of sorrowful developers to remind you that it has gone away.

2.10.2 Connecting to the buildmaster

If the buildslave cannot connect to the buildmaster, the reason should be described in the '`twistd.log`' logfile. Some common problems are an incorrect master hostname or port number, or a mistyped bot name or password. If the buildslave loses the connection to the master, it is supposed to attempt to reconnect with an exponentially-increasing backoff. Each attempt (and the time of the next attempt) will be logged. If you get impatient, just manually stop and re-start the buildslave.

When the buildmaster is restarted, all slaves will be disconnected, and will attempt to reconnect as usual. The reconnect time will depend upon how long the buildmaster is offline (i.e. how far up the exponential backoff curve the slaves have travelled). Again, `buildbot stop` *`BASEDIR`*`; buildbot start` *`BASEDIR`* will speed up the process.

2.10.3 Forcing Builds

From the buildmaster's main status web page, you can force a build to be run on your build slave if you set the option `allowForce=True` in your `master.cfg` file see Section 7.1.2 [Enabling the "Force Build" Button], page 107.

Figure out which column is for a builder that runs on your slave, click on that builder's name, and the page that comes up will have a "Force Build" button. Fill in the form, hit the button, and a moment later you should see your slave's '`twistd.log`' filling with commands being run. Using `pstree` or `top` should also reveal the cvs/make/gcc/etc processes being run by the buildslave. Note that the same web page should also show the '`admin`' and '`host`' information files that you configured earlier.

3 Concepts

This chapter defines some of the basic concepts that the Buildbot uses. You'll need to understand how the Buildbot sees the world to configure it properly.

3.1 Version Control Systems

These source trees come from a Version Control System of some kind. CVS and Subversion are two popular ones, but the Buildbot supports others. All VC systems have some notion of an upstream `repository` which acts as a server[1], from which clients can obtain source trees according to various parameters. The VC repository provides source trees of various projects, for different branches, and from various points in time. The first thing we have to do is to specify which source tree we want to get.

3.1.1 Generalizing VC Systems

For the purposes of the Buildbot, we will try to generalize all VC systems as having repositories that each provide sources for a variety of projects. Each project is defined as a directory tree with source files. The individual files may each have revisions, but we ignore that and treat the project as a whole as having a set of revisions (CVS is really the only VC system still in widespread use that has per-file revisions.. everything modern has moved to atomic tree-wide changesets). Each time someone commits a change to the project, a new revision becomes available. These revisions can be described by a tuple with two items: the first is a branch tag, and the second is some kind of revision stamp or timestamp. Complex projects may have multiple branch tags, but there is always a default branch. The timestamp may be an actual timestamp (such as the -D option to CVS), or it may be a monotonically-increasing transaction number (such as the change number used by SVN and P4, or the revision number used by Arch/Baz/Bazaar, or a labeled tag used in CVS)[2]. The SHA1 revision ID used by Monotone, Mercurial, and Git is also a kind of revision stamp, in that it specifies a unique copy of the source tree, as does a Darcs "context" file.

When we aren't intending to make any changes to the sources we check out (at least not any that need to be committed back upstream), there are two basic ways to use a VC system:

- Retrieve a specific set of source revisions: some tag or key is used to index this set, which is fixed and cannot be changed by subsequent developers committing new changes to the tree. Releases are built from tagged revisions like this, so that they can be rebuilt again later (probably with controlled modifications).
- Retrieve the latest sources along a specific branch: some tag is used to indicate which branch is to be used, but within that constraint we want to get the latest revisions.

Build personnel or CM staff typically use the first approach: the build that results is (ideally) completely specified by the two parameters given to the VC system: repository and revision tag. This gives QA and end-users something concrete to point at when reporting

[1] except Darcs, but since the Buildbot never modifies its local source tree we can ignore the fact that Darcs uses a less centralized model

[2] many VC systems provide more complexity than this: in particular the local views that P4 and ClearCase can assemble out of various source directories are more complex than we're prepared to take advantage of here

bugs. Release engineers are also reportedly fond of shipping code that can be traced back to a concise revision tag of some sort.

Developers are more likely to use the second approach: each morning the developer does an update to pull in the changes committed by the team over the last day. These builds are not easy to fully specify: it depends upon exactly when you did a checkout, and upon what local changes the developer has in their tree. Developers do not normally tag each build they produce, because there is usually significant overhead involved in creating these tags. Recreating the trees used by one of these builds can be a challenge. Some VC systems may provide implicit tags (like a revision number), while others may allow the use of timestamps to mean "the state of the tree at time X" as opposed to a tree-state that has been explicitly marked.

The Buildbot is designed to help developers, so it usually works in terms of *the latest* sources as opposed to specific tagged revisions. However, it would really prefer to build from reproducible source trees, so implicit revisions are used whenever possible.

3.1.2 Source Tree Specifications

So for the Buildbot's purposes we treat each VC system as a server which can take a list of specifications as input and produce a source tree as output. Some of these specifications are static: they are attributes of the builder and do not change over time. Others are more variable: each build will have a different value. The repository is changed over time by a sequence of Changes, each of which represents a single developer making changes to some set of files. These Changes are cumulative[3].

For normal builds, the Buildbot wants to get well-defined source trees that contain specific Changes, and exclude other Changes that may have occurred after the desired ones. We assume that the Changes arrive at the buildbot (through one of the mechanisms described in see Section 5.1 [Change Sources], page 50) in the same order in which they are committed to the repository. The Buildbot waits for the tree to become "stable" before initiating a build, for two reasons. The first is that developers frequently make multiple related commits in quick succession, even when the VC system provides ways to make atomic transactions involving multiple files at the same time. Running a build in the middle of these sets of changes would use an inconsistent set of source files, and is likely to fail (and is certain to be less useful than a build which uses the full set of changes). The tree-stable-timer is intended to avoid these useless builds that include some of the developer's changes but not all. The second reason is that some VC systems (i.e. CVS) do not provide repository-wide transaction numbers, so that timestamps are the only way to refer to a specific repository state. These timestamps may be somewhat ambiguous, due to processing and notification delays. By waiting until the tree has been stable for, say, 10 minutes, we can choose a timestamp from the middle of that period to use for our source checkout, and then be reasonably sure that any clock-skew errors will not cause the build to be performed on an inconsistent set of source files.

The Schedulers always use the tree-stable-timer, with a timeout that is configured to reflect a reasonable tradeoff between build latency and change frequency. When the VC system provides coherent repository-wide revision markers (such as Subversion's revision

[3] Monotone's *multiple heads* feature violates this assumption of cumulative Changes, but in most situations the changes don't occur frequently enough for this to be a significant problem

numbers, or in fact anything other than CVS's timestamps), the resulting Build is simply performed against a source tree defined by that revision marker. When the VC system does not provide this, a timestamp from the middle of the tree-stable period is used to generate the source tree[4].

3.1.3 How Different VC Systems Specify Sources

For CVS, the static specifications are `repository` and `module`. In addition to those, each build uses a timestamp (or omits the timestamp to mean `the latest`) and `branch tag` (which defaults to HEAD). These parameters collectively specify a set of sources from which a build may be performed.

Subversion (`http://subversion.tigris.org`) combines the repository, module, and branch into a single `Subversion URL` parameter. Within that scope, source checkouts can be specified by a numeric `revision number` (a repository-wide monotonically-increasing marker, such that each transaction that changes the repository is indexed by a different revision number), or a revision timestamp. When branches are used, the repository and module form a static `baseURL`, while each build has a `revision number` and a `branch` (which defaults to a statically-specified `defaultBranch`). The `baseURL` and `branch` are simply concatenated together to derive the `svnurl` to use for the checkout.

Perforce (`http://www.perforce.com/`) is similar. The server is specified through a `P4PORT` parameter. Module and branch are specified in a single depot path, and revisions are depot-wide. When branches are used, the `p4base` and `defaultBranch` are concatenated together to produce the depot path.

Arch (`http://wiki.gnuarch.org/`) and Bazaar (`http://bazaar.canonical.com/`) specify a repository by URL, as well as a `version` which is kind of like a branch name. Arch uses the word `archive` to represent the repository. Arch lets you push changes from one archive to another, removing the strict centralization required by CVS and SVN. It retains the distinction between repository and working directory that most other VC systems use. For complex multi-module directory structures, Arch has a built-in `build config` layer with which the checkout process has two steps. First, an initial bootstrap checkout is performed to retrieve a set of build-config files. Second, one of these files is used to figure out which archives/modules should be used to populate subdirectories of the initial checkout.

Builders which use Arch and Bazaar therefore have a static archive `url`, and a default "branch" (which is a string that specifies a complete category–branch–version triple). Each build can have its own branch (the category–branch–version string) to override the default, as well as a revision number (which is turned into a –patch-NN suffix when performing the checkout).

Bzr (`http://bazaar-vcs.org`) (which is a descendant of Arch/Bazaar, and is frequently referred to as "Bazaar") has the same sort of repository-vs-workspace model as Arch, but the repository data can either be stored inside the working directory or kept elsewhere (either on the same machine or on an entirely different machine). For the purposes of Buildbot (which never commits changes), the repository is specified with a URL and a revision number.

[4] this `checkoutDelay` defaults to half the tree-stable timer, but it can be overridden with an argument to the Source Step

The most common way to obtain read-only access to a bzr tree is via HTTP, simply by making the repository visible through a web server like Apache. Bzr can also use FTP and SFTP servers, if the buildslave process has sufficient privileges to access them. Higher performance can be obtained by running a special Bazaar-specific server. None of these matter to the buildbot: the repository URL just has to match the kind of server being used. The `repoURL` argument provides the location of the repository.

Branches are expressed as subdirectories of the main central repository, which means that if branches are being used, the BZR step is given a `baseURL` and `defaultBranch` instead of getting the `repoURL` argument.

Darcs (`http://darcs.net/`) doesn't really have the notion of a single master repository. Nor does it really have branches. In Darcs, each working directory is also a repository, and there are operations to push and pull patches from one of these `repositories` to another. For the Buildbot's purposes, all you need to do is specify the URL of a repository that you want to build from. The build slave will then pull the latest patches from that repository and build them. Multiple branches are implemented by using multiple repositories (possibly living on the same server).

Builders which use Darcs therefore have a static `repourl` which specifies the location of the repository. If branches are being used, the source Step is instead configured with a `baseURL` and a `defaultBranch`, and the two strings are simply concatenated together to obtain the repository's URL. Each build then has a specific branch which replaces `defaultBranch`, or just uses the default one. Instead of a revision number, each build can have a "context", which is a string that records all the patches that are present in a given tree (this is the output of `darcs changes --context`, and is considerably less concise than, e.g. Subversion's revision number, but the patch-reordering flexibility of Darcs makes it impossible to provide a shorter useful specification).

Mercurial (`http://selenic.com/mercurial`) is like Darcs, in that each branch is stored in a separate repository. The `repourl`, `baseURL`, and `defaultBranch` arguments are all handled the same way as with Darcs. The "revision", however, is the hash identifier returned by `hg identify`.

Git (`http://git.or.cz/`) also follows a decentralized model, and each repository can have several branches and tags. The source Step is configured with a static `repourl` which specifies the location of the repository. In addition, an optional `branch` parameter can be specified to check out code from a specific branch instead of the default "master" branch. The "revision" is specified as a SHA1 hash as returned by e.g. `git rev-parse`. No attempt is made to ensure that the specified revision is actually a subset of the specified branch.

Monotone (`http://code.monotone.ca/p/monotone`) follows the a decentralized model as well, where each repository can have several branches and tags. The source Step is configured with `repourl` which specifies the location of the repository, and `branch`, which is the default branch to use. The "revision" is given as a SHA1 hash that is returned by `mtn heads` among others. No attempt is made to ensure that the specified revision is a subset of the specified branch.

3.1.4 Attributes of Changes

Who

Each Change has a `who` attribute, which specifies which developer is responsible for the change. This is a string which comes from a namespace controlled by the VC repository. Frequently this means it is a username on the host which runs the repository, but not all VC systems require this (Arch, for example, uses a fully-qualified `Arch ID`, which looks like an email address, as does Darcs). Each StatusNotifier will map the `who` attribute into something appropriate for their particular means of communication: an email address, an IRC handle, etc.

Files

It also has a list of `files`, which are just the tree-relative filenames of any files that were added, deleted, or modified for this Change. These filenames are used by the `fileIsImportant` function (in the Scheduler) to decide whether it is worth triggering a new build or not, e.g. the function could use the following function to only run a build if a C file were checked in:

```
def has_C_files(change):
    for name in change.files:
        if name.endswith(".c"):
            return True
    return False
```

Certain BuildSteps can also use the list of changed files to run a more targeted series of tests, e.g. the `python_twisted.Trial` step can run just the unit tests that provide coverage for the modified .py files instead of running the full test suite.

Comments

The Change also has a `comments` attribute, which is a string containing any checkin comments.

Revision

Each Change can have a `revision` attribute, which describes how to get a tree with a specific state: a tree which includes this Change (and all that came before it) but none that come after it. If this information is unavailable, the `.revision` attribute will be `None`. These revisions are provided by the ChangeSource, and consumed by the `computeSourceRevision` method in the appropriate `source.Source` class.

'`CVS`' `revision` is an int, seconds since the epoch

'`SVN`' `revision` is an int, the changeset number (r%d)

'`Darcs`' `revision` is a large string, the output of `darcs changes --context`

'`Mercurial`'
`revision` is a short string (a hash ID), the output of `hg identify`

'`Arch/Bazaar`'
`revision` is the full revision ID (ending in –patch-%d)

'`P4`' `revision` is an int, the transaction number

'Git' `revision` is a short string (a SHA1 hash), the output of e.g. `git rev-parse`

'Monotone'
`revision` is a short string (a SHA1 hash), the output of e.g. `mtn heads`

Branches

The Change might also have a `branch` attribute. This indicates that all of the Change's files are in the same named branch. The Schedulers get to decide whether the branch should be built or not.

For VC systems like CVS, Arch, Monotone, and Git, the `branch` name is unrelated to the filename. (that is, the branch name and the filename inhabit unrelated namespaces). For SVN, branches are expressed as subdirectories of the repository, so the file's "svnurl" is a combination of some base URL, the branch name, and the filename within the branch. (In a sense, the branch name and the filename inhabit the same namespace). Darcs branches are subdirectories of a base URL just like SVN. Mercurial branches are the same as Darcs.

'CVS' branch='warner-newfeature', files=['src/foo.c']

'SVN' branch='branches/warner-newfeature', files=['src/foo.c']

'Darcs' branch='warner-newfeature', files=['src/foo.c']

'Mercurial'
branch='warner-newfeature', files=['src/foo.c']

'Arch/Bazaar'
branch='buildbot–usebranches–0', files=['buildbot/master.py']

'Git' branch='warner-newfeature', files=['src/foo.c']

'Monotone'
branch='warner-newfeature', files=['src/foo.c']

Build Properties

A Change may have one or more properties attached to it, usually specified through the Force Build form or see Section 8.3.1 [sendchange], page 127. Properties are discussed in detail in the see Section 3.7 [Build Properties], page 27 section.

Links

Finally, the Change might have a `links` list, which is intended to provide a list of URLs to a *viewcvs*-style web page that provides more detail for this Change, perhaps including the full file diffs.

3.2 Schedulers

Each Buildmaster has a set of `Scheduler` objects, each of which gets a copy of every incoming Change. The Schedulers are responsible for deciding when Builds should be run. Some Buildbot installations might have a single Scheduler, while others may have several, each for a different purpose.

For example, a "quick" scheduler might exist to give immediate feedback to developers, hoping to catch obvious problems in the code that can be detected quickly. These typically do not run the full test suite, nor do they run on a wide variety of platforms. They also usually do a VC update rather than performing a brand-new checkout each time. You could have a "quick" scheduler which used a 30 second timeout, and feeds a single "quick" Builder that uses a VC `mode='update'` setting.

A separate "full" scheduler would run more comprehensive tests a little while later, to catch more subtle problems. This scheduler would have a longer tree-stable-timer, maybe 30 minutes, and would feed multiple Builders (with a `mode=` of `'copy'`, `'clobber'`, or `'export'`).

The `tree-stable-timer` and `fileIsImportant` decisions are made by the Scheduler. Dependencies are also implemented here. Periodic builds (those which are run every N seconds rather than after new Changes arrive) are triggered by a special `Periodic` Scheduler subclass. The default Scheduler class can also be told to watch for specific branches, ignoring Changes on other branches. This may be useful if you have a trunk and a few release branches which should be tracked, but when you don't want to have the Buildbot pay attention to several dozen private user branches.

When the setup has multiple sources of Changes the `category` can be used for `Scheduler` objects to filter out a subset of the Changes. Note that not all change sources can attach a category.

Some Schedulers may trigger builds for other reasons, other than recent Changes. For example, a Scheduler subclass could connect to a remote buildmaster and watch for builds of a library to succeed before triggering a local build that uses that library.

Each Scheduler creates and submits `BuildSet` objects to the `BuildMaster`, which is then responsible for making sure the individual `BuildRequests` are delivered to the target `Builders`.

`Scheduler` instances are activated by placing them in the `c['schedulers']` list in the buildmaster config file. Each Scheduler has a unique name.

3.3 BuildSet

A `BuildSet` is the name given to a set of Builds that all compile/test the same version of the tree on multiple Builders. In general, all these component Builds will perform the same sequence of Steps, using the same source code, but on different platforms or against a different set of libraries.

The `BuildSet` is tracked as a single unit, which fails if any of the component Builds have failed, and therefore can succeed only if *all* of the component Builds have succeeded. There are two kinds of status notification messages that can be emitted for a BuildSet: the `firstFailure` type (which fires as soon as we know the BuildSet will fail), and the `Finished` type (which fires once the BuildSet has completely finished, regardless of whether the overall set passed or failed).

A `BuildSet` is created with a *source stamp* tuple of (branch, revision, changes, patch), some of which may be None, and a list of Builders on which it is to be run. They are then given to the BuildMaster, which is responsible for creating a separate `BuildRequest` for each Builder.

There are a couple of different likely values for the `SourceStamp`:

`(revision=None, changes=[CHANGES], patch=None)`
: This is a `SourceStamp` used when a series of Changes have triggered a build. The VC step will attempt to check out a tree that contains CHANGES (and any changes that occurred before CHANGES, but not any that occurred after them).

`(revision=None, changes=None, patch=None)`
: This builds the most recent code on the default branch. This is the sort of `SourceStamp` that would be used on a Build that was triggered by a user request, or a Periodic scheduler. It is also possible to configure the VC Source Step to always check out the latest sources rather than paying attention to the Changes in the SourceStamp, which will result in same behavior as this.

`(branch=BRANCH, revision=None, changes=None, patch=None)`
: This builds the most recent code on the given BRANCH. Again, this is generally triggered by a user request or Periodic build.

`(revision=REV, changes=None, patch=(LEVEL, DIFF, SUBDIR_ROOT))`
: This checks out the tree at the given revision REV, then applies a patch (using `patch -pLEVEL <DIFF`) from inside the relative directory SUBDIR_ROOT. Item SUBDIR_ROOT is optional and defaults to the builder working directory. The Section 8.2.3 [try], page 121 feature uses this kind of `SourceStamp`. If `patch` is None, the patching step is bypassed.

The buildmaster is responsible for turning the `BuildSet` into a set of `BuildRequest` objects and queueing them on the appropriate Builders.

3.4 BuildRequest

A `BuildRequest` is a request to build a specific set of sources on a single specific `Builder`. Each `Builder` runs the `BuildRequest` as soon as it can (i.e. when an associated buildslave becomes free). `BuildRequest`s are prioritized from oldest to newest, so when a buildslave becomes free, the `Builder` with the oldest `BuildRequest` is run.

The `BuildRequest` contains the `SourceStamp` specification. The actual process of running the build (the series of Steps that will be executed) is implemented by the `Build` object. In this future this might be changed, to have the `Build` define *what* gets built, and a separate `BuildProcess` (provided by the Builder) to define *how* it gets built.

`BuildRequest` is created with optional `Properties`. One of these, `owner`, is collected by the resultant `Build` and added to the set of *interested users* to which status notifications will be sent, depending on the configuration for each status object.

The `BuildRequest` may be mergeable with other compatible `BuildRequest`s. Builds that are triggered by incoming Changes will generally be mergeable. Builds that are triggered by user requests are generally not, unless they are multiple requests to build the *latest sources* of the same branch.

3.5 Builder

The Buildmaster runs a collection of Builders, each of which handles a single type of build (e.g. full versus quick), on one or more build slaves. Builders serve as a kind of queue for

a particular type of build. Each Builder gets a separate column in the waterfall display. In general, each Builder runs independently (although various kinds of interlocks can cause one Builder to have an effect on another).

Each `Builder` is a long-lived object which controls a sequence of Builds. Each Builder is created when the config file is first parsed, and lives forever (or rather until it is removed from the config file). It mediates the connections to the buildslaves that do all the work, and is responsible for creating the `Build` objects that decide *how* a build is performed (i.e., which steps are executed in what order).

Each `Builder` gets a unique name, and the path name of a directory where it gets to do all its work (there is a buildmaster-side directory for keeping status information, as well as a buildslave-side directory where the actual checkout/compile/test commands are executed). It also gets a `BuildFactory`, which is responsible for creating new `Build` instances: because the `Build` instance is what actually performs each build, choosing the `BuildFactory` is the way to specify what happens each time a build is done.

Each `Builder` is associated with one of more `BuildSlaves`. A `Builder` which is used to perform OS-X builds (as opposed to Linux or Solaris builds) should naturally be associated with an OS-X-based buildslave.

If multiple buildslaves are available for any given Builder, you will have some measure of redundancy: in case one slave goes offline, the others can still keep the Builder working. In addition, multiple buildslaves will allow multiple simultaneous builds for the same Builder, which might be useful if you have a lot of forced or "try" builds taking place.

If you use this feature, it is important to make sure that the buildslaves are all, in fact, capable of running the given build. The slave hosts should be configured similarly, otherwise you will spend a lot of time trying (unsuccessfully) to reproduce a failure that only occurs on some of the buildslaves and not the others. Different platforms, operating systems, versions of major programs or libraries, all these things mean you should use separate Builders.

3.6 Users

Buildbot has a somewhat limited awareness of *users*. It assumes the world consists of a set of developers, each of whom can be described by a couple of simple attributes. These developers make changes to the source code, causing builds which may succeed or fail.

Each developer is primarily known through the source control system. Each Change object that arrives is tagged with a `who` field that typically gives the account name (on the repository machine) of the user responsible for that change. This string is the primary key by which the User is known, and is displayed on the HTML status pages and in each Build's "blamelist".

To do more with the User than just refer to them, this username needs to be mapped into an address of some sort. The responsibility for this mapping is left up to the status module which needs the address. The core code knows nothing about email addresses or IRC nicknames, just user names.

3.6.1 Doing Things With Users

Each Change has a single User who is responsible for that Change. Most Builds have a set of Changes: the Build represents the first time these Changes have been built and tested

by the Buildbot. The build has a "blamelist" that consists of a simple union of the Users responsible for all the Build's Changes.

The Build provides (through the IBuildStatus interface) a list of Users who are "involved" in the build. For now this is equal to the blamelist, but in the future it will be expanded to include a "build sheriff" (a person who is "on duty" at that time and responsible for watching over all builds that occur during their shift), as well as per-module owners who simply want to keep watch over their domain (chosen by subdirectory or a regexp matched against the filenames pulled out of the Changes). The Involved Users are those who probably have an interest in the results of any given build.

In the future, Buildbot will acquire the concept of "Problems", which last longer than builds and have beginnings and ends. For example, a test case which passed in one build and then failed in the next is a Problem. The Problem lasts until the test case starts passing again, at which point the Problem is said to be "resolved".

If there appears to be a code change that went into the tree at the same time as the test started failing, that Change is marked as being resposible for the Problem, and the user who made the change is added to the Problem's "Guilty" list. In addition to this user, there may be others who share responsibility for the Problem (module owners, sponsoring developers). In addition to the Responsible Users, there may be a set of Interested Users, who take an interest in the fate of the Problem.

Problems therefore have sets of Users who may want to be kept aware of the condition of the problem as it changes over time. If configured, the Buildbot can pester everyone on the Responsible list with increasing harshness until the problem is resolved, with the most harshness reserved for the Guilty parties themselves. The Interested Users may merely be told when the problem starts and stops, as they are not actually responsible for fixing anything.

3.6.2 Email Addresses

The `buildbot.status.mail.MailNotifier` class (see Section 7.2 [MailNotifier], page 112) provides a status target which can send email about the results of each build. It accepts a static list of email addresses to which each message should be delivered, but it can also be configured to send mail to the Build's Interested Users. To do this, it needs a way to convert User names into email addresses.

For many VC systems, the User Name is actually an account name on the system which hosts the repository. As such, turning the name into an email address is a simple matter of appending "`@`repositoryhost.com". Some projects use other kinds of mappings (for example the preferred email address may be at "project.org" despite the repository host being named "cvs.project.org"), and some VC systems have full separation between the concept of a user and that of an account on the repository host (like Perforce). Some systems (like Arch) put a full contact email address in every change.

To convert these names to addresses, the MailNotifier uses an EmailLookup object. This provides a .getAddress method which accepts a name and (eventually) returns an address. The default `MailNotifier` module provides an EmailLookup which simply appends a static string, configurable when the notifier is created. To create more complex behaviors (perhaps using an LDAP lookup, or using "finger" on a central host to determine a preferred address for the developer), provide a different object as the `lookup` argument.

In the future, when the Problem mechanism has been set up, the Buildbot will need to send mail to arbitrary Users. It will do this by locating a MailNotifier-like object among all the buildmaster's status targets, and asking it to send messages to various Users. This means the User-to-address mapping only has to be set up once, in your MailNotifier, and every email message the buildbot emits will take advantage of it.

3.6.3 IRC Nicknames

Like MailNotifier, the `buildbot.status.words.IRC` class provides a status target which can announce the results of each build. It also provides an interactive interface by responding to online queries posted in the channel or sent as private messages.

In the future, the buildbot can be configured map User names to IRC nicknames, to watch for the recent presence of these nicknames, and to deliver build status messages to the interested parties. Like `MailNotifier` does for email addresses, the `IRC` object will have an `IRCLookup` which is responsible for nicknames. The mapping can be set up statically, or it can be updated by online users themselves (by claiming a username with some kind of "buildbot: i am user warner" commands).

Once the mapping is established, the rest of the buildbot can ask the `IRC` object to send messages to various users. It can report on the likelihood that the user saw the given message (based upon how long the user has been inactive on the channel), which might prompt the Problem Hassler logic to send them an email message instead.

3.6.4 Live Status Clients

The Buildbot also offers a PB-based status client interface which can display real-time build status in a GUI panel on the developer's desktop. This interface is normally anonymous, but it could be configured to let the buildmaster know *which* developer is using the status client. The status client could then be used as a message-delivery service, providing an alternative way to deliver low-latency high-interruption messages to the developer (like "hey, you broke the build").

3.7 Build Properties

Each build has a set of "Build Properties", which can be used by its BuildStep to modify their actions. These properties, in the form of key-value pairs, provide a general framework for dynamically altering the behavior of a build based on its circumstances.

Properties come from a number of places:

- global configuration – These properties apply to all builds.
- schedulers – A scheduler can specify properties available to all the builds it starts.
- changes – A change can have properties attached to it. These are usually specified through Force Build or sendchange.
- buildslaves – A buildslave can pass properties on to the builds it performs.
- builds – A build automatically sets a number of properties on itself.
- steps – Steps of a build can set properties that are available to subsequent steps. In particular, source steps set a number of properties.

Properties are very flexible, and can be used to implement all manner of functionality. Here are some examples:

Most Source steps record the revision that they checked out in the `got_revision` property. A later step could use this property to specify the name of a fully-built tarball, dropped in an easily-acessible directory for later testing.

Some projects want to perform nightly builds as well as in response to committed changes. Such a project would run two schedulers, both pointing to the same set of builders, but could provide an `is_nightly` property so that steps can distinguish the nightly builds, perhaps to run more resource-intensive tests.

Some projects have different build processes on different systems. Rather than create a build factory for each slave, the steps can use buildslave properties to identify the unique aspects of each slave and adapt the build process dynamically.

4 Configuration

The buildbot's behavior is defined by the "config file", which normally lives in the '`master.cfg`' file in the buildmaster's base directory (but this can be changed with an option to the `buildbot create-master` command). This file completely specifies which Builders are to be run, which slaves they should use, how Changes should be tracked, and where the status information is to be sent. The buildmaster's '`buildbot.tac`' file names the base directory; everything else comes from the config file.

A sample config file was installed for you when you created the buildmaster, but you will need to edit it before your buildbot will do anything useful.

This chapter gives an overview of the format of this file and the various sections in it. You will need to read the later chapters to understand how to fill in each section properly.

4.1 Config File Format

The config file is, fundamentally, just a piece of Python code which defines a dictionary named `BuildmasterConfig`, with a number of keys that are treated specially. You don't need to know Python to do basic configuration, though, you can just copy the syntax of the sample file. If you *are* comfortable writing Python code, however, you can use all the power of a full programming language to achieve more complicated configurations.

The `BuildmasterConfig` name is the only one which matters: all other names defined during the execution of the file are discarded. When parsing the config file, the Buildmaster generally compares the old configuration with the new one and performs the minimum set of actions necessary to bring the buildbot up to date: Builders which are not changed are left untouched, and Builders which are modified get to keep their old event history.

Basic Python syntax: comments start with a hash character ("#"), tuples are defined with `(parenthesis, pairs)`, arrays are defined with `[square, brackets]`, tuples and arrays are mostly interchangeable. Dictionaries (data structures which map "keys" to "values") are defined with curly braces: `{'key1': 'value1', 'key2': 'value2'}` . Function calls (and object instantiation) can use named parameters, like `w = html.Waterfall(http_port=8010)`.

The config file starts with a series of `import` statements, which make various kinds of Steps and Status targets available for later use. The main `BuildmasterConfig` dictionary is created, then it is populated with a variety of keys. These keys are broken roughly into the following sections, each of which is documented in the rest of this chapter:

- Project Definitions
- Change Sources / Schedulers
- Slaveport
- Buildslave Configuration
- Builders / Interlocks
- Status Targets
- Debug options

The config file can use a few names which are placed into its namespace:

`basedir` the base directory for the buildmaster. This string has not been expanded, so it may start with a tilde. It needs to be expanded before use. The config file is located in `os.path.expanduser(os.path.join(basedir, 'master.cfg'))`

4.2 Loading the Config File

The config file is only read at specific points in time. It is first read when the buildmaster is launched. Once it is running, there are various ways to ask it to reload the config file. If you are on the system hosting the buildmaster, you can send a `SIGHUP` signal to it: the `buildbot` tool has a shortcut for this:

```
buildbot reconfig BASEDIR
```

This command will show you all of the lines from '`twistd.log`' that relate to the reconfiguration. If there are any problems during the config-file reload, they will be displayed in these lines.

The debug tool (`buildbot debugclient --master HOST:PORT`) has a "Reload .cfg" button which will also trigger a reload. In the future, there will be other ways to accomplish this step (probably a password-protected button on the web page, as well as a privileged IRC command).

When reloading the config file, the buildmaster will endeavor to change as little as possible about the running system. For example, although old status targets may be shut down and new ones started up, any status targets that were not changed since the last time the config file was read will be left running and untouched. Likewise any Builders which have not been changed will be left running. If a Builder is modified (say, the build process is changed) while a Build is currently running, that Build will keep running with the old process until it completes. Any previously queued Builds (or Builds which get queued after the reconfig) will use the new process.

4.3 Testing the Config File

To verify that the config file is well-formed and contains no deprecated or invalid elements, use the "checkconfig" command, passing it either a master directory or a config file.

```
% buildbot checkconfig master.cfg
Config file is good!
# or
% buildbot checkconfig /tmp/masterdir
Config file is good!
```

If the config file has deprecated features (perhaps because you've upgraded the buildmaster and need to update the config file to match), they will be announced by checkconfig. In this case, the config file will work, but you should really remove the deprecated items and use the recommended replacements instead:

```
% buildbot checkconfig master.cfg
/usr/lib/python2.4/site-packages/buildbot/master.py:559: DeprecationWarning: c['sou
deprecated as of 0.7.6 and will be removed by 0.8.0 . Please use c['change_source']
  warnings.warn(m, DeprecationWarning)
Config file is good!
```

If the config file is simply broken, that will be caught too:

```
% buildbot checkconfig master.cfg
Traceback (most recent call last):
  File "/usr/lib/python2.4/site-packages/buildbot/scripts/runner.py", line 834, in
    ConfigLoader(configFile)
  File "/usr/lib/python2.4/site-packages/buildbot/scripts/checkconfig.py", line 31,
    self.loadConfig(configFile)
  File "/usr/lib/python2.4/site-packages/buildbot/master.py", line 480, in loadConf
    exec f in localDict
  File "/home/warner/BuildBot/master/foolscap/master.cfg", line 90, in ?
    c[bogus] = "stuff"
NameError: name 'bogus' is not defined
```

4.4 Defining the Project

There are a couple of basic settings that you use to tell the buildbot what project it is working on. This information is used by status reporters to let users find out more about the codebase being exercised by this particular Buildbot installation.

```
c['projectName'] = "Buildbot"
c['projectURL'] = "http://buildbot.sourceforge.net/"
c['buildbotURL'] = "http://localhost:8010/"
```

`projectName` is a short string will be used to describe the project that this buildbot is working on. For example, it is used as the title of the waterfall HTML page.

`projectURL` is a string that gives a URL for the project as a whole. HTML status displays will show `projectName` as a link to `projectURL`, to provide a link from buildbot HTML pages to your project's home page.

The `buildbotURL` string should point to the location where the buildbot's internal web server (usually the `html.Waterfall` page) is visible. This typically uses the port number set when you create the `Waterfall` object: the buildbot needs your help to figure out a suitable externally-visible host name.

When status notices are sent to users (either by email or over IRC), `buildbotURL` will be used to create a URL to the specific build or problem that they are being notified about. It will also be made available to queriers (over IRC) who want to find out where to get more information about this buildbot.

The `logCompressionLimit` enables compression of build logs on disk for logs that are bigger than the given size, or disables that completely if given `False`. The default value is 4k, which should be a reasonable default on most file systems. This setting has no impact on status plugins, and merely affects the required disk space on the master for build logs.

The `logCompressionMethod` controls what type of compression is used for build logs. The default is 'bz2', the other valid option is 'gz'. 'bz2' offers better compression at the expense of more CPU time.

The `logMaxSize` parameter sets an upper limit (in bytes) to how large logs from an individual build step can be. The default value is None, meaning no upper limit to the log size. Any output exceeding `logMaxSize` will be truncated, and a message to this effect will be added to the log's HEADER channel.

If `logMaxSize` is set, and the output from a step exceeds the maximum, the `logMaxTailSize` parameter controls how much of the end of the build log will be kept.

The effect of setting this parameter is that the log will contain the first `logMaxSize` bytes and the last `logMaxTailSize` bytes of output. Don't set this value too high, as the the tail of the log is kept in memory.

4.5 Change Sources and Schedulers

The `c['change_source']` key is the ChangeSource instance[1] that defines how the buildmaster learns about source code changes. More information about what goes here is available in See Chapter 5 [Getting Source Code Changes], page 50.

```
from buildbot.changes.pb import PBChangeSource
c['change_source'] = PBChangeSource()
```

(note: in buildbot-0.7.5 and earlier, this key was named `c['sources']`, and required a list. `c['sources']` is deprecated as of buildbot-0.7.6 and is scheduled to be removed in a future release).

The `c['changeHorizon']` key determines how many changes the master will keep a record of. One place these changes are displayed is on the waterfall page. This parameter defaults to 0, which means keep all changes indefinitely.

`c['schedulers']` is a list of Scheduler instances, each of which causes builds to be started on a particular set of Builders. The two basic Scheduler classes you are likely to start with are `Scheduler` and `Periodic`, but you can write a customized subclass to implement more complicated build scheduling.

Scheduler arguments should always be specified by name (as keyword arguments), to allow for future expansion:

```
sched = Scheduler(name="quick", builderNames=['lin', 'win'])
```

All schedulers have several arguments in common:

`name`
: Each Scheduler must have a unique name. This is used in status displays, and is also available in the build property `scheduler`.

`builderNames`
: This is the set of builders which this scheduler should trigger, specified as a list of names (strings).

`properties`
: This is a dictionary specifying properties that will be transmitted to all builds started by this scheduler.

Here is a brief catalog of the available Scheduler types. All these Schedulers are classes in `buildbot.scheduler`, and the docstrings there are the best source of documentation on the arguments taken by each one.

[1] To be precise, it is an object or a list of objects which all implement the `buildbot.interfaces.IChangeSource` Interface. It is unusual to have multiple ChangeSources, so this key accepts either a single ChangeSource or a sequence of them.

4.5.1 Scheduler Scheduler

This is the original and still most popular Scheduler class. It follows exactly one branch, and starts a configurable tree-stable-timer after each change on that branch. When the timer expires, it starts a build on some set of Builders. The Scheduler accepts a `fileIsImportant` function which can be used to ignore some Changes if they do not affect any "important" files.

The arguments to this scheduler are:

`name`

`builderNames`

`properties`

`branch` This Scheduler will pay attention to a single branch, ignoring Changes that occur on other branches. Setting `branch` equal to the special value of `None` means it should only pay attention to the default branch. Note that `None` is a keyword, not a string, so you want to use `None` and not `"None"`.

`treeStableTimer`
The Scheduler will wait for this many seconds before starting the build. If new changes are made during this interval, the timer will be restarted, so really the build will be started after a change and then after this many seconds of inactivity.

`fileIsImportant`
A callable which takes one argument, a Change instance, and returns `True` if the change is worth building, and `False` if it is not. Unimportant Changes are accumulated until the build is triggered by an important change. The default value of None means that all Changes are important.

`categories`
A list of categories of changes that this scheduler will respond to. If this is specified, then any non-matching changes are ignored.

Example:

```
from buildbot import scheduler
quick = scheduler.Scheduler(name="quick",
                    branch=None,
                    treeStableTimer=60,
                    builderNames=["quick-linux", "quick-netbsd"])
full = scheduler.Scheduler(name="full",
                    branch=None,
                    treeStableTimer=5*60,
                    builderNames=["full-linux", "full-netbsd", "full-OSX"])
c['schedulers'] = [quick, full]
```

In this example, the two "quick" builders are triggered 60 seconds after the tree has been changed. The "full" builds do not run quite so quickly (they wait 5 minutes), so hopefully if the quick builds fail due to a missing file or really simple typo, the developer can discover and fix the problem before the full builds are started. Both Schedulers only pay attention to the default branch: any changes on other branches are ignored by these Schedulers. Each Scheduler triggers a different set of Builders, referenced by name.

4.5.2 AnyBranchScheduler

This scheduler uses a tree-stable-timer like the default one, but follows multiple branches at once. Each branch gets a separate timer.

The arguments to this scheduler are:

`name`

`builderNames`
`properties`
`branches` This Scheduler will pay attention to any number of branches, ignoring Changes that occur on other branches. Branches are specified just as for the `Scheduler` class.

`treeStableTimer`
The Scheduler will wait for this many seconds before starting the build. If new changes are made during this interval, the timer will be restarted, so really the build will be started after a change and then after this many seconds of inactivity.

`fileIsImportant`
A callable which takes one argument, a Change instance, and returns `True` if the change is worth building, and `False` if it is not. Unimportant Changes are accumulated until the build is triggered by an important change. The default value of None means that all Changes are important.

`categories`
A list of categories of changes that this scheduler will respond to. If this is specified, then any non-matching changes are ignored.

4.5.3 Dependent Scheduler

It is common to wind up with one kind of build which should only be performed if the same source code was successfully handled by some other kind of build first. An example might be a packaging step: you might only want to produce .deb or RPM packages from a tree that was known to compile successfully and pass all unit tests. You could put the packaging step in the same Build as the compile and testing steps, but there might be other reasons to not do this (in particular you might have several Builders worth of compiles/tests, but only wish to do the packaging once). Another example is if you want to skip the "full" builds after a failing "quick" build of the same source code. Or, if one Build creates a product (like a compiled library) that is used by some other Builder, you'd want to make sure the consuming Build is run *after* the producing one.

You can use "Dependencies" to express this relationship to the Buildbot. There is a special kind of Scheduler named `scheduler.Dependent` that will watch an "upstream" Scheduler for builds to complete successfully (on all of its Builders). Each time that happens, the same source code (i.e. the same `SourceStamp`) will be used to start a new set of builds, on a different set of Builders. This "downstream" scheduler doesn't pay attention to Changes at all. It only pays attention to the upstream scheduler.

If the build fails on any of the Builders in the upstream set, the downstream builds will not fire. Note that, for SourceStamps generated by a ChangeSource, the `revision` is None,

meaning HEAD. If any changes are committed between the time the upstream scheduler begins its build and the time the dependent scheduler begins its build, then those changes will be included in the downstream build. See the see Section 4.5.7 [Triggerable Scheduler], page 37 for a more flexible dependency mechanism that can avoid this problem.

The keyword arguments to this scheduler are:

`name`

`builderNames`
`properties`
`upstream` The upstream scheduler to watch. Note that this is an "instance", not the name of the scheduler.

Example:

```
from buildbot import scheduler
tests = scheduler.Scheduler("just-tests", None, 5*60,
                            ["full-linux", "full-netbsd", "full-OSX"])
package = scheduler.Dependent(name="build-package",
                              upstream=tests, # <- no quotes!
                              builderNames=["make-tarball", "make-deb", "make-rpm"]
c['schedulers'] = [tests, package]
```

4.5.4 Periodic Scheduler

This simple scheduler just triggers a build every N seconds.

The arguments to this scheduler are:

`name`

`builderNames`
`properties`
`periodicBuildTimer`
The time, in seconds, after which to start a build.

Example:

```
from buildbot import scheduler
nightly = scheduler.Periodic(name="nightly",
                builderNames=["full-solaris"],
                periodicBuildTimer=24*60*60)
c['schedulers'] = [nightly]
```

The Scheduler in this example just runs the full solaris build once per day. Note that this Scheduler only lets you control the time between builds, not the absolute time-of-day of each Build, so this could easily wind up a "daily" or "every afternoon" scheduler depending upon when it was first activated.

4.5.5 Nightly Scheduler

This is highly configurable periodic build scheduler, which triggers a build at particular times of day, week, month, or year. The configuration syntax is very similar to the well-known `crontab` format, in which you provide values for minute, hour, day, and month (some of which can be wildcards), and a build is triggered whenever the current time matches the

given constraints. This can run a build every night, every morning, every weekend, alternate Thursdays, on your boss's birthday, etc.

Pass some subset of `minute`, `hour`, `dayOfMonth`, `month`, and `dayOfWeek`; each may be a single number or a list of valid values. The builds will be triggered whenever the current time matches these values. Wildcards are represented by a '*' string. All fields default to a wildcard except 'minute', so with no fields this defaults to a build every hour, on the hour. The full list of parameters is:

`name`

`builderNames`
`properties`
`branch` The branch to build, just as for `Scheduler`.

`minute` The minute of the hour on which to start the build. This defaults to 0, meaning an hourly build.

`hour` The hour of the day on which to start the build, in 24-hour notation. This defaults to *, meaning every hour.

`month` The month in which to start the build, with January = 1. This defaults to *, meaning every month.

`dayOfWeek`
The day of the week to start a build, with Monday = 0. This defauls to *, meaning every day of the week.

`onlyIfChanged`
If this is true, then builds will not be scheduled at the designated time unless the source has changed since the previous build.

For example, the following master.cfg clause will cause a build to be started every night at 3:00am:

```
s = scheduler.Nightly(name='nightly',
        builderNames=['builder1', 'builder2'],
        hour=3,
        minute=0)
```

This scheduler will perform a build each monday morning at 6:23am and again at 8:23am, but only if someone has committed code in the interim:

```
s = scheduler.Nightly(name='BeforeWork',
         builderNames=['builder1'],
         dayOfWeek=0,
         hour=[6,8],
         minute=23,
         onlyIfChanged=True)
```

The following runs a build every two hours, using Python's `range` function:

```
s = Nightly(name='every2hours',
        builderNames=['builder1'],
        hour=range(0, 24, 2))
```

Finally, this example will run only on December 24th:

```
s = Nightly(name='SleighPreflightCheck',
        builderNames=['flying_circuits', 'radar'],
        month=12,
        dayOfMonth=24,
        hour=12,
        minute=0)
```

4.5.6 Try Schedulers

This scheduler allows developers to use the `buildbot try` command to trigger builds of code they have not yet committed. See Section 8.2.3 [try], page 121 for complete details.

Two implementations are available: `Try_Jobdir` and `Try_Userpass`. The former monitors a job directory, specified by the `jobdir` parameter, while the latter listens for PB connections on a specific `port`, and authenticates against `userport`.

4.5.7 Triggerable Scheduler

The `Triggerable` scheduler waits to be triggered by a Trigger step (see Section 6.1.10 [Triggering Schedulers], page 87) in another build. That step can optionally wait for the scheduler's builds to complete. This provides two advantages over Dependent schedulers. First, the same scheduler can be triggered from multiple builds. Second, the ability to wait for a Triggerable's builds to complete provides a form of "subroutine call", where one or more builds can "call" a scheduler to perform some work for them, perhaps on other buildslaves.

The parameters are just the basics:

`name`

`builderNames`
`properties`

This class is only useful in conjunction with the `Trigger` step. Here is a fully-worked example:

```
from buildbot import scheduler
from buildbot.process import factory
from buildbot.steps import trigger

checkin = scheduler.Scheduler(name="checkin",
            branch=None,
            treeStableTimer=5*60,
            builderNames=["checkin"])
nightly = scheduler.Nightly(name='nightly',
            builderNames=['nightly'],
            hour=3,
            minute=0)

mktarball = scheduler.Triggerable(name="mktarball",
                builderNames=["mktarball"])
build = scheduler.Triggerable(name="build-all-platforms",
                builderNames=["build-all-platforms"])
```

```
test = scheduler.Triggerable(name="distributed-test",
                builderNames=["distributed-test"])
package = scheduler.Triggerable(name="package-all-platforms",
                builderNames=["package-all-platforms"])

c['schedulers'] = [mktarball, checkin, nightly, build, test, package]

# on checkin, make a tarball, build it, and test it
checkin_factory = factory.BuildFactory()
checkin_factory.addStep(trigger.Trigger(schedulerNames=['mktarball'],
                                        waitForFinish=True))
checkin_factory.addStep(trigger.Trigger(schedulerNames=['build-all-platforms'],
                                    waitForFinish=True))
checkin_factory.addStep(trigger.Trigger(schedulerNames=['distributed-test'],
                                    waitForFinish=True))

# and every night, make a tarball, build it, and package it
nightly_factory = factory.BuildFactory()
nightly_factory.addStep(trigger.Trigger(schedulerNames=['mktarball'],
                                        waitForFinish=True))
nightly_factory.addStep(trigger.Trigger(schedulerNames=['build-all-platforms'],
                                    waitForFinish=True))
nightly_factory.addStep(trigger.Trigger(schedulerNames=['package-all-platforms'],
                                      waitForFinish=True))
```

4.6 Merging BuildRequests

By default, buildbot merges BuildRequests that have the compatible SourceStamps. This behaviour can be customized with the `c['mergeRequests']` configuration key. This key specifies a function which is called with three arguments: a `Builder` and two `BuildRequest` objects. It should return true if the requests can be merged. For example:

```
def mergeRequests(builder, req1, req2):
    """Don't merge buildrequest at all"""
    return False
c['mergeRequests'] = mergeRequests
```

In many cases, the details of the SourceStamps and BuildRequests are important. In this example, only BuildRequests with the same "reason" are merged; thus developers forcing builds for different reasons will see distinct builds.

```
def mergeRequests(builder, req1, req2):
    if req1.source.canBeMergedWith(req2.source) and  req1.reason == req2.reason:
       return True
    return False
c['mergeRequests'] = mergeRequests
```

4.7 Prioritizing Builders

By default, buildbot will attempt to start builds on builders in order from the builder with the oldest pending request to the newest. This behaviour can be customized with the `c['prioritizeBuilders']` configuration key. This key specifies a function which is called with two arguments: a `BuildMaster` and a list of `Builder` objects. It should return a list of `Builder` objects in the desired order. It may also remove items from the list if builds should not be started on those builders.

```
def prioritizeBuilders(buildmaster, builders):
    """Prioritize builders.  'finalRelease' builds have the highest
    priority, so they should be built before running tests, or
    creating builds."""
    builderPriorities = {
        "finalRelease": 0,
        "test": 1,
        "build": 2,
    }
    builders.sort(key=lambda b: builderPriorities.get(b.name, 0))
    return builders

c['prioritizeBuilders'] = prioritizeBuilders
```

4.8 Setting the slaveport

The buildmaster will listen on a TCP port of your choosing for connections from buildslaves. It can also use this port for connections from remote Change Sources, status clients, and debug tools. This port should be visible to the outside world, and you'll need to tell your buildslave admins about your choice.

It does not matter which port you pick, as long it is externally visible, however you should probably use something larger than 1024, since most operating systems don't allow non-root processes to bind to low-numbered ports. If your buildmaster is behind a firewall or a NAT box of some sort, you may have to configure your firewall to permit inbound connections to this port.

```
c['slavePortnum'] = 10000
```

`c['slavePortnum']` is a *strports* specification string, defined in the `twisted.application.strports` module (try `pydoc twisted.application.strports` to get documentation on the format). This means that you can have the buildmaster listen on a localhost-only port by doing:

```
c['slavePortnum'] = "tcp:10000:interface=127.0.0.1"
```

This might be useful if you only run buildslaves on the same machine, and they are all configured to contact the buildmaster at `localhost:10000`.

4.9 Buildslave Specifiers

The `c['slaves']` key is a list of known buildslaves. In the common case, each buildslave is defined by an instance of the BuildSlave class. It represents a standard, manually started machine that will try to connect to the buildbot master as a slave. Contrast these with the

"on-demand" latent buildslaves, such as the Amazon Web Service Elastic Compute Cloud latent buildslave discussed below.

The BuildSlave class is instantiated with two values: (slavename, slavepassword). These are the same two values that need to be provided to the buildslave administrator when they create the buildslave.

The slavenames must be unique, of course. The password exists to prevent evildoers from interfering with the buildbot by inserting their own (broken) buildslaves into the system and thus displacing the real ones.

Buildslaves with an unrecognized slavename or a non-matching password will be rejected when they attempt to connect, and a message describing the problem will be put in the log file (see Section 2.7 [Logfiles], page 14).

```
from buildbot.buildslave import BuildSlave
c['slaves'] = [BuildSlave('bot-solaris', 'solarispasswd')
               BuildSlave('bot-bsd', 'bsdpasswd')
              ]
```

`BuildSlave` objects can also be created with an optional `properties` argument, a dictionary specifying properties that will be available to any builds performed on this slave. For example:

```
from buildbot.buildslave import BuildSlave
c['slaves'] = [BuildSlave('bot-solaris', 'solarispasswd',
                      properties={'os':'solaris'}),
              ]
```

The `BuildSlave` constructor can also take an optional `max_builds` parameter to limit the number of builds that it will execute simultaneously:

```
from buildbot.buildslave import BuildSlave
c['slaves'] = [BuildSlave("bot-linux", "linuxpassword", max_builds=2)]
```

Historical note: in buildbot-0.7.5 and earlier, the `c['bots']` key was used instead, and it took a list of (name, password) tuples. This key is accepted for backwards compatibility, but is deprecated as of 0.7.6 and will go away in some future release.

4.9.1 When Buildslaves Go Missing

Sometimes, the buildslaves go away. One very common reason for this is when the buildslave process is started once (manually) and left running, but then later the machine reboots and the process is not automatically restarted.

If you'd like to have the administrator of the buildslave (or other people) be notified by email when the buildslave has been missing for too long, just add the `notify_on_missing=` argument to the `BuildSlave` definition:

```
c['slaves'] = [BuildSlave('bot-solaris', 'solarispasswd',
                          notify_on_missing="bob@example.com"),
              ]
```

By default, this will send email when the buildslave has been disconnected for more than one hour. Only one email per connection-loss event will be sent. To change the timeout, use `missing_timeout=` and give it a number of seconds (the default is 3600).

You can have the buildmaster send email to multiple recipients: just provide a list of addresses instead of a single one:

```
c['slaves'] = [BuildSlave('bot-solaris', 'solarispasswd',
                          notify_on_missing=["bob@example.com",
                                             "alice@example.org"],
                          missing_timeout=300, # notify after 5 minutes
                          ),
               ]
```

The email sent this way will use a MailNotifier (see Section 7.2 [MailNotifier], page 112) status target, if one is configured. This provides a way for you to control the "from" address of the email, as well as the relayhost (aka "smarthost") to use as an SMTP server. If no MailNotifier is configured on this buildmaster, the buildslave-missing emails will be sent using a default configuration.

Note that if you want to have a MailNotifier for buildslave-missing emails but not for regular build emails, just create one with builders=[], as follows:

```
from buildbot.status import mail
m = mail.MailNotifier(fromaddr="buildbot@localhost", builders=[],
                      relayhost="smtp.example.org")
c['status'].append(m)
c['slaves'] = [BuildSlave('bot-solaris', 'solarispasswd',
                          notify_on_missing="bob@example.com"),
               ]
```

4.10 On-Demand ("Latent") Buildslaves

The standard buildbot model has slaves started manually. The previous section described how to configure the master for this approach.

Another approach is to let the buildbot master start slaves when builds are ready, on-demand. Thanks to services such as Amazon Web Services' Elastic Compute Cloud ("AWS EC2"), this is relatively easy to set up, and can be very useful for some situations.

The buildslaves that are started on-demand are called "latent" buildslaves. As of this writing, buildbot ships with an abstract base class for building latent buildslaves, and a concrete implementation for AWS EC2.

4.10.1 Amazon Web Services Elastic Compute Cloud ("AWS EC2")

AWS EC2 is a web service that allows you to start virtual machines in an Amazon data center. Please see their website for details, incuding costs. Using the AWS EC2 latent buildslaves involves getting an EC2 account with AWS and setting up payment; customizing one or more EC2 machine images ("AMIs") on your desired operating system(s) and publishing them (privately if needed); and configuring the buildbot master to know how to start your customized images for "substantiating" your latent slaves.

4.10.1.1 Get an AWS EC2 Account

To start off, to use the AWS EC2 latent buildslave, you need to get an AWS developer account and sign up for EC2. These instructions may help you get started:

- Go to http://aws.amazon.com/ and click to "Sign Up Now" for an AWS account.
- Once you are logged into your account, you need to sign up for EC2. Instructions for how to do this have changed over time because Amazon changes their website, so the best advice is to hunt for it. After signing up for EC2, it may say it wants you to upload an x.509 cert. You will need this to create images (see below) but it is not technically necessary for the buildbot master configuration.
- You must enter a valid credit card before you will be able to use EC2. Do that under 'Payment Method'.
- Make sure you're signed up for EC2 by going to 'Your Account'->'Account Activity' and verifying EC2 is listed.

4.10.1.2 Create an AMI

Now you need to create an AMI and configure the master. You may need to run through this cycle a few times to get it working, but these instructions should get you started.

Creating an AMI is out of the scope of this document. The EC2 Getting Started Guide is a good resource for this task. Here are a few additional hints.

- When an instance of the image starts, it needs to automatically start a buildbot slave that connects to your master (to create a buildbot slave, see Section 2.5 [Creating a buildslave], page 10; to make a daemon, see Section 2.6 [Launching the daemons], page 13).
- You may want to make an instance of the buildbot slave, configure it as a standard buildslave in the master (i.e., not as a latent slave), and test and debug it that way before you turn it into an AMI and convert to a latent slave in the master.

4.10.1.3 Configure the Master with an EC2LatentBuildSlave

Now let's assume you have an AMI that should work with the EC2LatentBuildSlave. It's now time to set up your buildbot master configuration.

You will need some information from your AWS account: the "Access Key Id" and the "Secret Access Key". If you've built the AMI yourself, you probably already are familiar with these values. If you have not, and someone has given you access to an AMI, these hints may help you find the necessary values:

- While logged into your AWS account, find the "Access Identifiers" link (either on the left, or via "Your Account" -> "Access Identifiers".
- On the page, you'll see alphanumeric values for "Your Access Key Id:" and "Your Secret Access Key:". Make a note of these. Later on, we'll call the first one your "identifier" and the second one your "secret_identifier."

When creating an EC2LatentBuildSlave in the buildbot master configuration, the first three arguments are required. The name and password are the first two arguments, and work the same as with normal buildslaves. The next argument specifies the type of the EC2 virtual machine (available options as of this writing include "m1.small", "m1.large", 'm1.xlarge", "c1.medium", and "c1.xlarge"; see the EC2 documentation for descriptions of these machines).

Here is the simplest example of configuring an EC2 latent buildslave. It specifies all necessary remaining values explicitly in the instantiation.

```
from buildbot.ec2buildslave import EC2LatentBuildSlave
c['slaves'] = [EC2LatentBuildSlave('bot1', 'sekrit', 'm1.large',
                                   ami='ami-12345',
                                   identifier='publickey',
                                   secret_identifier='privatekey'
                                   )]
```

The "ami" argument specifies the AMI that the master should start. The "identifier" argument specifies the AWS "Access Key Id," and the "secret_identifier" specifies the AWS "Secret Access Key." Both the AMI and the account information can be specified in alternate ways.

Note that whoever has your identifier and secret_identifier values can request AWS work charged to your account, so these values need to be carefully protected. Another way to specify these access keys is to put them in a separate file. You can then make the access privileges stricter for this separate file, and potentially let more people read your main configuration file.

By default, you can make an .ec2 directory in the home folder of the user running the buildbot master. In that directory, create a file called aws_id. The first line of that file should be your access key id; the second line should be your secret access key id. Then you can instantiate the build slave as follows.

```
from buildbot.ec2buildslave import EC2LatentBuildSlave
c['slaves'] = [EC2LatentBuildSlave('bot1', 'sekrit', 'm1.large',
                                   ami='ami-12345')]
```

If you want to put the key information in another file, use the "aws_id_file_path" initialization argument.

Previous examples used a particular AMI. If the Buildbot master will be deployed in a process-controlled environment, it may be convenient to specify the AMI more flexibly. Rather than specifying an individual AMI, specify one or two AMI filters.

In all cases, the AMI that sorts last by its location (the S3 bucket and manifest name) will be preferred.

One available filter is to specify the acceptable AMI owners, by AWS account number (the 12 digit number, usually rendered in AWS with hyphens like "1234-5678-9012", should be entered as in integer).

```
from buildbot.ec2buildslave import EC2LatentBuildSlave
bot1 = EC2LatentBuildSlave('bot1', 'sekrit', 'm1.large',
                           valid_ami_owners=[11111111111,
                                             22222222222],
                           identifier='publickey',
                           secret_identifier='privatekey'
                           )
```

The other available filter is to provide a regular expression string that will be matched against each AMI's location (the S3 bucket and manifest name).

```
from buildbot.ec2buildslave import EC2LatentBuildSlave
bot1 = EC2LatentBuildSlave(
    'bot1', 'sekrit', 'm1.large',
```

```
    valid_ami_location_regex=r'buildbot\-.*/image.manifest.xml',
    identifier='publickey', secret_identifier='privatekey')
```

The regular expression can specify a group, which will be preferred for the sorting. Only the first group is used; subsequent groups are ignored.

```
from buildbot.ec2buildslave import EC2LatentBuildSlave
bot1 = EC2LatentBuildSlave(
    'bot1', 'sekrit', 'm1.large',
    valid_ami_location_regex=r'buildbot\-.*\-(.*)/image.manifest.xml',
    identifier='publickey', secret_identifier='privatekey')
```

If the group can be cast to an integer, it will be. This allows 10 to sort after 1, for instance.

```
from buildbot.ec2buildslave import EC2LatentBuildSlave
bot1 = EC2LatentBuildSlave(
    'bot1', 'sekrit', 'm1.large',
    valid_ami_location_regex=r'buildbot\-.*\-(\d+)/image.manifest.xml',
    identifier='publickey', secret_identifier='privatekey')
```

In addition to using the password as a handshake between the master and the slave, you may want to use a firewall to assert that only machines from a specific IP can connect as slaves. This is possible with AWS EC2 by using the Elastic IP feature. To configure, generate a Elastic IP in AWS, and then specify it in your configuration using the "elastic_ip" argument.

```
from buildbot.ec2buildslave import EC2LatentBuildSlave
c['slaves'] = [EC2LatentBuildSlave('bot1', 'sekrit', 'm1.large',
                                   'ami-12345',
                                   identifier='publickey',
                                   secret_identifier='privatekey',
                                   elastic_ip='208.77.188.166'
                                   )]
```

The EC2LatentBuildSlave supports all other configuration from the standard BuildSlave. The "missing_timeout" and "notify_on_missing" specify how long to wait for an EC2 instance to attach before considering the attempt to have failed, and email addresses to alert, respectively. "missing_timeout" defaults to 20 minutes.

The "build_wait_timeout" allows you to specify how long an EC2LatentBuildSlave should wait after a build for another build before it shuts down the EC2 instance. It defaults to 10 minutes.

"keypair_name" and "security_name" allow you to specify different names for these AWS EC2 values. They both default to "latent_buildbot_slave".

4.10.2 Dangers with Latent Buildslaves

Any latent build slave that interacts with a for-fee service, such as the EC2LatentBuildSlave, brings significant risks. As already identified, the configuraton will need access to account information that, if obtained by a criminal, can be used to charge services to your account. Also, bugs in the buildbot software may lead to unnecessary charges. In particular, if the master neglects to shut down an instance for some reason, a virtual machine may be running

unnecessarily, charging against your account. Manual and/or automatic (e.g. nagios with a plugin using a library like boto) double-checking may be appropriate.

A comparitively trivial note is that currently if two instances try to attach to the same latent buildslave, it is likely that the system will become confused. This should not occur, unless, for instance, you configure a normal build slave to connect with the authentication of a latent buildbot. If the situation occurs, stop all attached instances and restart the master.

4.10.3 Writing New Latent Buildslaves

Writing a new latent buildslave should only require subclassing `buildbot.buildslave.AbstractLatentBui` and implementing start_instance and stop_instance.

```
def start_instance(self):
    # responsible for starting instance that will try to connect with this
    # master. Should return deferred. Problems should use an errback. The
    # callback value can be None, or can be an iterable of short strings to
    # include in the "substantiate success" status message, such as
    # identifying the instance that started.
    raise NotImplementedError

def stop_instance(self, fast=False):
    # responsible for shutting down instance. Return a deferred. If `fast`,
    # we're trying to shut the master down, so callback as soon as is safe.
    # Callback value is ignored.
    raise NotImplementedError
```

See `buildbot.ec2buildslave.EC2LatentBuildSlave` for an example, or see the test example `buildbot.test_slaves.FakeLatentBuildSlave`.

4.11 Defining Global Properties

The `'properties'` configuration key defines a dictionary of properties that will be available to all builds started by the buildmaster:

```
c['properties'] = {
    'Widget-version' : '1.2',
    'release-stage' : 'alpha'
}
```

4.12 Defining Builders

The `c['builders']` key is a list of objects giving configuration for the Builders. For more information, See Section 3.5 [Builder], page 24. The class definition for the builder configuration is in `buildbot.config`. In the configuration file, its use looks like:

```
from buildbot.config import BuilderConfig
c['builders'] = [
    BuilderConfig(name='quick', slavenames=['bot1', 'bot2'], factory=f_quick),
    BuilderConfig(name='thorough', slavename='bot1', factory=f_thorough),
]
```

The constructor takes the following keyword arguments:

`name` This specifies the Builder's name, which is used in status reports.

`slavename`
`slavenames`
These arguments specify the buildslave or buildslaves that will be used by this Builder. All slaves names must appear in the `c['slaves']` list. Each buildslave can accomodate multiple Builders. The `slavenames` parameter can be a list of names, while `slavename` can specify only one slave.

`factory` This is a `buildbot.process.factory.BuildFactory` instance which controls how the build is performed. Full details appear in their own section, See Chapter 6 [Build Process], page 66. Parameters like the location of the CVS repository and the compile-time options used for the build are generally provided as arguments to the factory's constructor.

Other optional keys may be set on each Builder:

`builddir` Specifies the name of a subdirectory (under the base directory) in which everything related to this builder will be placed on the buildmaster. This holds build status information. If not set, defaults to `name` with some characters escaped. Each builder must have a unique build directory.

`slavebuilddir`
Specifies the name of a subdirectory (under the base directory) in which everything related to this builder will be placed on the buildslave. This is where checkouts, compiles, and tests are run. If not set, defaults to `builddir`. If a slave is connected to multiple builders that share the same `slavebuilddir`, make sure the slave is set to run one build at a time or ensure this is fine to run multiple builds from the same directory simultaneously.

`category` If provided, this is a string that identifies a category for the builder to be a part of. Status clients can limit themselves to a subset of the available categories. A common use for this is to add new builders to your setup (for a new module, or for a new buildslave) that do not work correctly yet and allow you to integrate them with the active builders. You can put these new builders in a test category, make your main status clients ignore them, and have only private status clients pick them up. As soon as they work, you can move them over to the active category.

`nextSlave`
If provided, this is a function that controls which slave will be assigned future jobs. The function is passed two arguments, the `Builder` object which is assigning a new job, and a list of `BuildSlave` objects. The function should return one of the `BuildSlave` objects, or `None` if none of the available slaves should be used.

`nextBuild`
If provided, this is a function that controls which build request will be handled next. The function is passed two arguments, the `Builder` object which is assigning a new job, and a list of `BuildRequest` objects of pending builds. The function should return one of the `BuildRequest` objects, or `None` if none of the pending builds should be started.

`locks` This argument specifies a list of locks that apply to this builder; See Section 6.2 [Interlocks], page 96.

`env` A Builder may be given a dictionary of environment variables in this parameter. The variables are used in see Section 6.1.4 [ShellCommand], page 78 steps in builds created by this builder. The environment variables will override anything in the buildslave's environment. Variables passed directly to a `ShellCommand` will override variables of the same name passed to the Builder.

For example, if you a pool of identical slaves it is often easier to manage variables like PATH from Buildbot rather than manually editing it inside of the slaves' environment.

```
f = factory.BuildFactory
f.addStep(ShellCommand(
              command=['bash', './configure']))
f.addStep(Compile())

c['builders'] = [
  BuilderConfig(name='test', factory=f,
        slavenames=['slave1', 'slave2', 'slave3', 'slave4'],
        env={'PATH': '/opt/local/bin:/opt/app/bin:/usr/local/bin:/usr/bin
]
```

4.13 Defining Status Targets

The Buildmaster has a variety of ways to present build status to various users. Each such delivery method is a "Status Target" object in the configuration's `status` list. To add status targets, you just append more objects to this list:

```
c['status'] = []

from buildbot.status import html
c['status'].append(html.Waterfall(http_port=8010))

from buildbot.status import mail
m = mail.MailNotifier(fromaddr="buildbot@localhost",
                      extraRecipients=["builds@lists.example.com"],
                      sendToInterestedUsers=False)
c['status'].append(m)

from buildbot.status import words
c['status'].append(words.IRC(host="irc.example.com", nick="bb",
                             channels=["#example"]))
```

Status delivery has its own chapter, See Chapter 7 [Status Delivery], page 105, in which all the built-in status targets are documented.

4.14 Limiting Memory and Disk Usage

Buildbot stores historical information on disk in the form of "Pickle" files and compressed logfiles. In a large installation, these can quickly consume disk space, yet in many cases

developers never consult this historical information. Four configuration parameters control the "pruning" of various data, shown here with their default values:

```
c['buildHorizon'] = 100
c['eventHorizon'] = 50
c['logHorizon'] = 40

c['buildCacheSize'] = 15
```

The `buildHorizon` specifies the minimum number of builds for each builder which should be kept on disk. The `eventHorizon` specifies the minumum number of events to keep – events mostly describe connections and disconnections of slaves, and are seldom helpful to developers. The `logHorizon` gives the minimum number of builds for which logs should be maintained; this parameter must be less than `buildHorizon`. Builds older than `logHorizon` but not older than `buildHorizon` will maintain their overall status and the status of each step, but the logfiles will be deleted.

Finally, the `buildCacheSize` gives the number of builds for each builder which are cached in memory. This number should be larger than the number of builds required for commonly-used status displays (the waterfall or grid views), so that those displays do not miss the cache on a refresh.

4.15 Debug options

If you set `c['debugPassword']`, then you can connect to the buildmaster with the diagnostic tool launched by `buildbot debugclient MASTER:PORT`. From this tool, you can reload the config file, manually force builds, and inject changes, which may be useful for testing your buildmaster without actually commiting changes to your repository (or before you have the Change Sources set up). The debug tool uses the same port number as the slaves do: `c['slavePortnum']`, and is authenticated with this password.

```
c['debugPassword'] = "debugpassword"
```

If you set `c['manhole']` to an instance of one of the classes in `buildbot.manhole`, you can telnet or ssh into the buildmaster and get an interactive Python shell, which may be useful for debugging buildbot internals. It is probably only useful for buildbot developers. It exposes full access to the buildmaster's account (including the ability to modify and delete files), so it should not be enabled with a weak or easily guessable password.

There are three separate `Manhole` classes. Two of them use SSH, one uses unencrypted telnet. Two of them use a username+password combination to grant access, one of them uses an SSH-style '`authorized_keys`' file which contains a list of ssh public keys.

`manhole.AuthorizedKeysManhole`
: You construct this with the name of a file that contains one SSH public key per line, just like '`~/.ssh/authorized_keys`'. If you provide a non-absolute filename, it will be interpreted relative to the buildmaster's base directory.

`manhole.PasswordManhole`
: This one accepts SSH connections but asks for a username and password when authenticating. It accepts only one such pair.

`manhole.TelnetManhole`

This accepts regular unencrypted telnet connections, and asks for a username/password pair before providing access. Because this username/password is transmitted in the clear, and because Manhole access to the buildmaster is equivalent to granting full shell privileges to both the buildmaster and all the buildslaves (and to all accounts which then run code produced by the buildslaves), it is highly recommended that you use one of the SSH manholes instead.

```
# some examples:
from buildbot import manhole
c['manhole'] = manhole.AuthorizedKeysManhole(1234, "authorized_keys")
c['manhole'] = manhole.PasswordManhole(1234, "alice", "mysecretpassword")
c['manhole'] = manhole.TelnetManhole(1234, "bob", "snoop_my_password_please")
```

The `Manhole` instance can be configured to listen on a specific port. You may wish to have this listening port bind to the loopback interface (sometimes known as "lo0", "localhost", or 127.0.0.1) to restrict access to clients which are running on the same host.

```
from buildbot.manhole import PasswordManhole
c['manhole'] = PasswordManhole("tcp:9999:interface=127.0.0.1","admin","passwd")
```

To have the `Manhole` listen on all interfaces, use `"tcp:9999"` or simply 9999. This port specification uses `twisted.application.strports`, so you can make it listen on SSL or even UNIX-domain sockets if you want.

Note that using any Manhole requires that the TwistedConch package be installed, and that you be using Twisted version 2.0 or later.

The buildmaster's SSH server will use a different host key than the normal sshd running on a typical unix host. This will cause the ssh client to complain about a "host key mismatch", because it does not realize there are two separate servers running on the same host. To avoid this, use a clause like the following in your '`.ssh/config`' file:

```
Host remotehost-buildbot
 HostName remotehost
 HostKeyAlias remotehost-buildbot
 Port 9999
 # use 'user' if you use PasswordManhole and your name is not 'admin'.
 # if you use AuthorizedKeysManhole, this probably doesn't matter.
 User admin
```

5 Getting Source Code Changes

The most common way to use the Buildbot is centered around the idea of `Source Trees`: a directory tree filled with source code of some form which can be compiled and/or tested. Some projects use languages that don't involve any compilation step: nevertheless there may be a `build` phase where files are copied or rearranged into a form that is suitable for installation. Some projects do not have unit tests, and the Buildbot is merely helping to make sure that the sources can compile correctly. But in all of these cases, the thing-being-tested is a single source tree.

A Version Control System mantains a source tree, and tells the buildmaster when it changes. The first step of each Build is typically to acquire a copy of some version of this tree.

This chapter describes how the Buildbot learns about what Changes have occurred. For more information on VC systems and Changes, see Section 3.1 [Version Control Systems], page 17.

5.1 Change Sources

Each Buildmaster watches a single source tree. Changes can be provided by a variety of ChangeSource types, however any given project will typically have only a single ChangeSource active. This section provides a description of all available ChangeSource types and explains how to set up each of them.

There are a variety of ChangeSources available, some of which are meant to be used in conjunction with other tools to deliver Change events from the VC repository to the buildmaster.

- CVSToys This ChangeSource opens a TCP connection from the buildmaster to a waiting FreshCVS daemon that lives on the repository machine, and subscribes to hear about Changes.
- MaildirSource This one watches a local maildir-format inbox for email sent out by the repository when a change is made. When a message arrives, it is parsed to create the Change object. A variety of parsing functions are available to accomodate different email-sending tools.
- PBChangeSource This ChangeSource listens on a local TCP socket for inbound connections from a separate tool. Usually, this tool would be run on the VC repository machine in a commit hook. It is expected to connect to the TCP socket and send a Change message over the network connection. The `buildbot sendchange` command is one example of a tool that knows how to send these messages, so you can write a commit script for your VC system that calls it to deliver the Change. There are other tools in the contrib/ directory that use the same protocol.

As a quick guide, here is a list of VC systems and the ChangeSources that might be useful with them. All of these ChangeSources are in the `buildbot.changes` module.

`CVS`

- freshcvs.FreshCVSSource (connected via TCP to the freshcvs daemon)
- mail.FCMaildirSource (watching for email sent by a freshcvs daemon)

- mail.BonsaiMaildirSource (watching for email sent by Bonsai)
- mail.SyncmailMaildirSource (watching for email sent by syncmail)
- pb.PBChangeSource (listening for connections from `buildbot sendchange` run in a loginfo script)
- pb.PBChangeSource (listening for connections from a long-running `contrib/viewcvspoll.py` polling process which examines the ViewCVS database directly

`SVN`

- pb.PBChangeSource (listening for connections from `contrib/svn_buildbot.py` run in a postcommit script)
- pb.PBChangeSource (listening for connections from a long-running `contrib/svn_watcher.py` or `contrib/svnpoller.py` polling process
- mail.SVNCommitEmailMaildirSource (watching for email sent by commit-email.pl)
- svnpoller.SVNPoller (polling the SVN repository)

`Darcs`

- pb.PBChangeSource (listening for connections from `contrib/darcs_buildbot.py` in a commit script

`Mercurial`

- pb.PBChangeSource (listening for connections from `contrib/hg_buildbot.py` run in an 'incoming' hook)
- pb.PBChangeSource (listening for connections from `buildbot/changes/hgbuildbot.py` run as an in-process 'changegroup' hook)

`Arch/Bazaar`

- pb.PBChangeSource (listening for connections from `contrib/arch_buildbot.py` run in a commit hook)

`Bzr (the newer Bazaar)`

- pb.PBChangeSource (listening for connections from `contrib/bzr_buildbot.py` run in a post-change-branch-tip or commit hook)
- `contrib/bzr_buildbot.py`'s BzrPoller (polling the Bzr repository)

`Git`

- pb.PBChangeSource (listening for connections from `contrib/git_buildbot.py` run in the post-receive hook)

`Monotone`

- pb.PBChangeSource (listening for connections from `contrib/monotone-buildbot-nofication.py` (found in the monotone source), run as a monotone hook)

All VC systems can be driven by a PBChangeSource and the `buildbot sendchange` tool run from some form of commit script. If you write an email parsing function, they can also all be driven by a suitable `MaildirSource`.

5.2 Choosing ChangeSources

The `master.cfg` configuration file has a dictionary key named `BuildmasterConfig['change_source']`, which holds the active `IChangeSource` object. The config file will typically create an object from one of the classes described below and stuff it into this key.

Each buildmaster typically has just a single ChangeSource, since it is only watching a single source tree. But if, for some reason, you need multiple sources, just set `c['change_source']` to a list of ChangeSources.. it will accept that too.

```
s = FreshCVSSourceNewcred(host="host", port=4519,
                          user="alice", passwd="secret",
                          prefix="Twisted")
BuildmasterConfig['change_source'] = [s]
```

Each source tree has a nominal `top`. Each Change has a list of filenames, which are all relative to this top location. The ChangeSource is responsible for doing whatever is necessary to accomplish this. Most sources have a `prefix` argument: a partial pathname which is stripped from the front of all filenames provided to that `ChangeSource`. Files which are outside this sub-tree are ignored by the changesource: it does not generate Changes for those files.

5.3 CVSToys - PBService

The CVSToys (`http://purl.net/net/CVSToys`) package provides a server which runs on the machine that hosts the CVS repository it watches. It has a variety of ways to distribute commit notifications, and offers a flexible regexp-based way to filter out uninteresting changes. One of the notification options is named `PBService` and works by listening on a TCP port for clients. These clients subscribe to hear about commit notifications.

The buildmaster has a CVSToys-compatible `PBService` client built in. There are two versions of it, one for old versions of CVSToys (1.0.9 and earlier) which used the `oldcred` authentication framework, and one for newer versions (1.0.10 and later) which use `newcred`. Both are classes in the `buildbot.changes.freshcvs` package.

`FreshCVSSourceNewcred` objects are created with the following parameters:

'`host and port`'
: these specify where the CVSToys server can be reached

'`user and passwd`'
: these specify the login information for the CVSToys server (`freshcvs`). These must match the server's values, which are defined in the `freshCfg` configuration file (which lives in the CVSROOT directory of the repository).

'`prefix`'
: this is the prefix to be found and stripped from filenames delivered by the CVSToys server. Most projects live in sub-directories of the main repository, as siblings of the CVSROOT sub-directory, so typically this prefix is set to that top sub-directory name.

Example

To set up the freshCVS server, add a statement like the following to your '`freshCfg`' file:

```
pb = ConfigurationSet([
    (None, None, None, PBService(userpass=('foo', 'bar'), port=4519)),
    ])
```

This will announce all changes to a client which connects to port 4519 using a username of 'foo' and a password of 'bar'.

Then add a clause like this to your buildmaster's 'master.cfg':

```
BuildmasterConfig['change_source'] = FreshCVSSource("cvs.example.com", 4519,
                                                    "foo", "bar",
                                                    prefix="glib/")
```

where "cvs.example.com" is the host that is running the FreshCVS daemon, and "glib" is the top-level directory (relative to the repository's root) where all your source code lives. Most projects keep one or more projects in the same repository (along with CVSROOT/ to hold admin files like loginfo and freshCfg); the prefix= argument tells the buildmaster to ignore everything outside that directory, and to strip that common prefix from all pathnames it handles.

5.4 Mail-parsing ChangeSources

Many projects publish information about changes to their source tree by sending an email message out to a mailing list, frequently named PROJECT-commits or PROJECT-changes. Each message usually contains a description of the change (who made the change, which files were affected) and sometimes a copy of the diff. Humans can subscribe to this list to stay informed about what's happening to the source tree.

The Buildbot can also be subscribed to a -commits mailing list, and can trigger builds in response to Changes that it hears about. The buildmaster admin needs to arrange for these email messages to arrive in a place where the buildmaster can find them, and configure the buildmaster to parse the messages correctly. Once that is in place, the email parser will create Change objects and deliver them to the Schedulers (see see Section 4.5 [Change Sources and Schedulers], page 32) just like any other ChangeSource.

There are two components to setting up an email-based ChangeSource. The first is to route the email messages to the buildmaster, which is done by dropping them into a "maildir". The second is to actually parse the messages, which is highly dependent upon the tool that was used to create them. Each VC system has a collection of favorite change-emailing tools, and each has a slightly different format, so each has a different parsing function. There is a separate ChangeSource variant for each parsing function.

Once you've chosen a maildir location and a parsing function, create the change source and put it in `c['change_source']`:

```
from buildbot.changes.mail import SyncmailMaildirSource
c['change_source'] = SyncmailMaildirSource("~/maildir-buildbot",
                                           prefix="/trunk/")
```

5.4.1 Subscribing the Buildmaster

The recommended way to install the buildbot is to create a dedicated account for the buildmaster. If you do this, the account will probably have a distinct email address (perhaps `buildmaster@example.org`). Then just arrange for this account's email to be delivered to a suitable maildir (described in the next section).

If the buildbot does not have its own account, "extension addresses" can be used to distinguish between email intended for the buildmaster and email intended for the rest of the account. In most modern MTAs, the e.g. `foo@example.org` account has control over every email address at example.org which begins with "foo", such that email addressed to `account-foo@example.org` can be delivered to a different destination than `account-bar@example.org`. qmail does this by using separate .qmail files for the two destinations ('`.qmail-foo`' and '`.qmail-bar`', with '`.qmail`' controlling the base address and '`.qmail-default`' controlling all other extensions). Other MTAs have similar mechanisms.

Thus you can assign an extension address like `foo-buildmaster@example.org` to the buildmaster, and retain `foo@example.org` for your own use.

5.4.2 Using Maildirs

A "maildir" is a simple directory structure originally developed for qmail that allows safe atomic update without locking. Create a base directory with three subdirectories: "new", "tmp", and "cur". When messages arrive, they are put into a uniquely-named file (using pids, timestamps, and random numbers) in "tmp". When the file is complete, it is atomically renamed into "new". Eventually the buildmaster notices the file in "new", reads and parses the contents, then moves it into "cur". A cronjob can be used to delete files in "cur" at leisure.

Maildirs are frequently created with the `maildirmake` tool, but a simple `mkdir -p ~/MAILDIR/{cur,new,tmp}` is pretty much equivalent.

Many modern MTAs can deliver directly to maildirs. The usual .forward or .procmailrc syntax is to name the base directory with a trailing slash, so something like `~/MAILDIR/` . qmail and postfix are maildir-capable MTAs, and procmail is a maildir-capable MDA (Mail Delivery Agent).

For MTAs which cannot put files into maildirs directly, the "safecat" tool can be executed from a .forward file to accomplish the same thing.

The Buildmaster uses the linux DNotify facility to receive immediate notification when the maildir's "new" directory has changed. When this facility is not available, it polls the directory for new messages, every 10 seconds by default.

5.4.3 Parsing Email Change Messages

The second component to setting up an email-based ChangeSource is to parse the actual notices. This is highly dependent upon the VC system and commit script in use.

A couple of common tools used to create these change emails are:

'`CVS`'

'`CVSToys MailNotifier`'
Section 5.4.3.1 [FCMaildirSource], page 55

'`Bonsai notification`'
Section 5.4.3.3 [BonsaiMaildirSource], page 56

'`syncmail`'
Section 5.4.3.2 [SyncmailMaildirSource], page 55

'`SVN`'

‘`svnmailer`’
http://opensource.perlig.de/en/svnmailer/

‘`commit-email.pl`’
Section 5.4.3.4 [SVNCommitEmailMaildirSource], page 56

‘`Bzr`’

‘`Launchpad`’
Section 5.4.3.5 [BzrLaunchpadEmailMaildirSource], page 56

‘`Mercurial`’

‘`NotifyExtension`’
http://www.selenic.com/mercurial/wiki/index.cgi/NotifyExtension

‘`Git`’

‘`post-receive-email`’
http://git.kernel.org/?p=git/git.git;a=blob;f=contrib/hooks/post-receive-email;hb=HEAD

The following sections describe the parsers available for each of these tools.

Most of these parsers accept a `prefix=` argument, which is used to limit the set of files that the buildmaster pays attention to. This is most useful for systems like CVS and SVN which put multiple projects in a single repository (or use repository names to indicate branches). Each filename that appears in the email is tested against the prefix: if the filename does not start with the prefix, the file is ignored. If the filename *does* start with the prefix, that prefix is stripped from the filename before any further processing is done. Thus the prefix usually ends with a slash.

5.4.3.1 FCMaildirSource

http://twistedmatrix.com/users/acapnotic/wares/code/CVSToys/

This parser works with the CVSToys `MailNotification` action, which will send email to a list of recipients for each commit. This tends to work better than using `/bin/mail` from within the CVSROOT/loginfo file directly, as CVSToys will batch together all files changed during the same CVS invocation, and can provide more information (like creating a ViewCVS URL for each file changed).

The Buildbot's `FCMaildirSource` knows for to parse these CVSToys messages and turn them into Change objects. It can be given two parameters: the directory name of the maildir root, and the prefix to strip.

```
from buildbot.changes.mail import FCMaildirSource
c['change_source'] = FCMaildirSource("~/maildir-buildbot")
```

5.4.3.2 SyncmailMaildirSource

http://sourceforge.net/projects/cvs-syncmail

`SyncmailMaildirSource` knows how to parse the message format used by the CVS “syncmail” script.

```
from buildbot.changes.mail import SyncmailMaildirSource
c['change_source'] = SyncmailMaildirSource("~/maildir-buildbot")
```

5.4.3.3 BonsaiMaildirSource

http://www.mozilla.org/bonsai.html

`BonsaiMaildirSource` parses messages sent out by Bonsai, the CVS tree-management system built by Mozilla.

```
from buildbot.changes.mail import BonsaiMaildirSource
c['change_source'] = BonsaiMaildirSource("~/maildir-buildbot")
```

5.4.3.4 SVNCommitEmailMaildirSource

`SVNCommitEmailMaildirSource` parses message sent out by the `commit-email.pl` script, which is included in the Subversion distribution.

It does not currently handle branches: all of the Change objects that it creates will be associated with the default (i.e. trunk) branch.

```
from buildbot.changes.mail import SVNCommitEmailMaildirSource
c['change_source'] = SVNCommitEmailMaildirSource("~/maildir-buildbot")
```

5.4.3.5 BzrLaunchpadEmailMaildirSource

`BzrLaunchpadEmailMaildirSource` parses the mails that are sent to addresses that subscribe to branch revision notifications for a bzr branch hosted on Launchpad.

The branch name defaults to `lp:<Launchpad path>`. For example `lp:~maria-captains/maria/5.1`.

If only a single branch is used, the default branch name can be changed by setting `defaultBranch`.

For multiple branches, pass a dictionary as the value of the `branchMap` option to map specific repository paths to specific branch names (see example below). The leading `lp:` prefix of the path is optional.

The `prefix` option is not supported (it is silently ignored). Use the `branchMap` and `defaultBranch` instead to assign changes to branches (and just do not subscribe the buildbot to branches that are not of interest).

The revision number is obtained from the email text. The bzr revision id is not available in the mails sent by Launchpad. However, it is possible to set the bzr `append_revisions_only` option for public shared repositories to avoid new pushes of merges changing the meaning of old revision numbers.

```
from buildbot.changes.mail import BzrLaunchpadEmailMaildirSource
bm = { 'lp:~maria-captains/maria/5.1' : '5.1', 'lp:~maria-captains/maria/6.0' : '6.0' }
c['change_source'] = BzrLaunchpadEmailMaildirSource("~/maildir-buildbot", branchMap = bm
```

5.5 PBChangeSource

The last kind of ChangeSource actually listens on a TCP port for clients to connect and push change notices *into* the Buildmaster. This is used by the built-in `buildbot sendchange` notification tool, as well as the VC-specific '`contrib/svn_buildbot.py`', '`contrib/arch_buildbot.py`', '`contrib/hg_buildbot.py`' tools, and the `buildbot.changes.hgbuildbot` hook. These tools are run by the repository (in a commit hook script), and connect to the buildmaster directly each time a file is

comitted. This is also useful for creating new kinds of change sources that work on a `push` model instead of some kind of subscription scheme, for example a script which is run out of an email .forward file.

This ChangeSource can be configured to listen on its own TCP port, or it can share the port that the buildmaster is already using for the buildslaves to connect. (This is possible because the `PBChangeSource` uses the same protocol as the buildslaves, and they can be distinguished by the `username` attribute used when the initial connection is established). It might be useful to have it listen on a different port if, for example, you wanted to establish different firewall rules for that port. You could allow only the SVN repository machine access to the `PBChangeSource` port, while allowing only the buildslave machines access to the slave port. Or you could just expose one port and run everything over it. *Note: this feature is not yet implemented, the PBChangeSource will always share the slave port and will always have a* `user` *name of* `change`*, and a passwd of* `changepw`*. These limitations will be removed in the future..*

The `PBChangeSource` is created with the following arguments. All are optional.

'`port`' which port to listen on. If `None` (which is the default), it shares the port used for buildslave connections. *Not Implemented, always set to* `None`.

'`user and passwd`'
The user/passwd account information that the client program must use to connect. Defaults to `change` and `changepw`. *Not Implemented,* `user` *is currently always set to* `change`*,* `passwd` *is always set to* `changepw`.

'`prefix`' The prefix to be found and stripped from filenames delivered over the connection. Any filenames which do not start with this prefix will be removed. If all the filenames in a given Change are removed, the that whole Change will be dropped. This string should probably end with a directory separator.

This is useful for changes coming from version control systems that represent branches as parent directories within the repository (like SVN and Perforce). Use a prefix of 'trunk/' or 'project/branches/foobranch/' to only follow one branch and to get correct tree-relative filenames. Without a prefix, the PBChangeSource will probably deliver Changes with filenames like '`trunk/foo.c`' instead of just '`foo.c`'. Of course this also depends upon the tool sending the Changes in (like `buildbot sendchange`) and what filenames it is delivering: that tool may be filtering and stripping prefixes at the sending end.

5.6 P4Source

The `P4Source` periodically polls a Perforce (`http://www.perforce.com/`) depot for changes. It accepts the following arguments:

'`p4base`' The base depot path to watch, without the trailing '/...'.

'`p4port`' The Perforce server to connect to (as host:port).

'`p4user`' The Perforce user.

'`p4passwd`'
The Perforce password.

'`p4bin`' An optional string parameter. Specify the location of the perforce command line binary (p4). You only need to do this if the perforce binary is not in the path of the buildbot user. Defaults to "p4".

'`split_file`'
A function that maps a pathname, without the leading `p4base`, to a (branch, filename) tuple. The default just returns (None, branchfile), which effectively disables branch support. You should supply a function which understands your repository structure.

'`pollinterval`'
How often to poll, in seconds. Defaults to 600 (10 minutes).

'`histmax`' The maximum number of changes to inspect at a time. If more than this number occur since the last poll, older changes will be silently ignored.

Example

This configuration uses the `P4PORT`, `P4USER`, and `P4PASSWD` specified in the buildmaster's environment. It watches a project in which the branch name is simply the next path component, and the file is all path components after.

```
from buildbot.changes import p4poller
s = p4poller.P4Source(p4base='//depot/project/',
                      split_file=lambda branchfile: branchfile.split('/',1),
                     )
c['change_source'] = s
```

5.7 BonsaiPoller

The `BonsaiPoller` periodically polls a Bonsai server. This is a CGI script accessed through a web server that provides information about a CVS tree, for example the Mozilla bonsai server at `http://bonsai.mozilla.org`. Bonsai servers are usable by both humans and machines. In this case, the buildbot's change source forms a query which asks about any files in the specified branch which have changed since the last query.

Please take a look at the BonsaiPoller docstring for details about the arguments it accepts.

5.8 SVNPoller

The `buildbot.changes.svnpoller.SVNPoller` is a ChangeSource which periodically polls a Subversion (`http://subversion.tigris.org/`) repository for new revisions, by running the `svn log` command in a subshell. It can watch a single branch or multiple branches.

`SVNPoller` accepts the following arguments:

`svnurl` The base URL path to watch, like `svn://svn.twistedmatrix.com/svn/Twisted/trunk`, or `http://divmod.org/svn/Divmod/`, or even `file:///home/svn/Repository/ProjectA/bra`
This must include the access scheme, the location of the repository (both the hostname for remote ones, and any additional directory names necessary to get to the repository), and the sub-path within the repository's virtual filesystem for the project and branch of interest.

The `SVNPoller` will only pay attention to files inside the subdirectory specified by the complete svnurl.

`split_file`
: A function to convert pathnames into (branch, relative_pathname) tuples. Use this to explain your repository's branch-naming policy to `SVNPoller`. This function must accept a single string and return a two-entry tuple. There are a few utility functions in `buildbot.changes.svnpoller` that can be used as a `split_file` function, see below for details.

  The default value always returns (None, path), which indicates that all files are on the trunk.

  Subclasses of `SVNPoller` can override the `split_file` method instead of using the `split_file=` argument.

`svnuser`
: An optional string parameter. If set, the `--user` argument will be added to all `svn` commands. Use this if you have to authenticate to the svn server before you can do `svn info` or `svn log` commands.

`svnpasswd`
: Like `svnuser`, this will cause a `--password` argument to be passed to all svn commands.

`pollinterval`
: How often to poll, in seconds. Defaults to 600 (checking once every 10 minutes). Lower this if you want the buildbot to notice changes faster, raise it if you want to reduce the network and CPU load on your svn server. Please be considerate of public SVN repositories by using a large interval when polling them.

`histmax`
: The maximum number of changes to inspect at a time. Every POLLINTERVAL seconds, the `SVNPoller` asks for the last HISTMAX changes and looks through them for any ones it does not already know about. If more than HISTMAX revisions have been committed since the last poll, older changes will be silently ignored. Larger values of histmax will cause more time and memory to be consumed on each poll attempt. `histmax` defaults to 100.

`svnbin`
: This controls the `svn` executable to use. If subversion is installed in a weird place on your system (outside of the buildmaster's `$PATH`), use this to tell `SVNPoller` where to find it. The default value of "svn" will almost always be sufficient.

`revlinktmpl`
: This parameter allows a link to be provided for each revision (for example, to websvn or viewvc). These links appear anywhere changes are shown, such as on build or change pages. The proper form for this parameter is an URL with the portion that will substitute for a revision number replaced by "%s". For example, `'http://myserver/websvn/revision.php?rev=%s'` could be used to cause revision links to be created to a websvn repository viewer.

Branches

Each source file that is tracked by a Subversion repository has a fully-qualified SVN URL in the following form: (REPOURL)(PROJECT-plus-BRANCH)(FILEPATH). When you

create the `SVNPoller`, you give it a `svnurl` value that includes all of the REPOURL and possibly some portion of the PROJECT-plus-BRANCH string. The `SVNPoller` is responsible for producing Changes that contain a branch name and a FILEPATH (which is relative to the top of a checked-out tree). The details of how these strings are split up depend upon how your repository names its branches.

PROJECT/BRANCHNAME/FILEPATH repositories

One common layout is to have all the various projects that share a repository get a single top-level directory each. Then under a given project's directory, you get two subdirectories, one named "trunk" and another named "branches". Under "branches" you have a bunch of other directories, one per branch, with names like "1.5.x" and "testing". It is also common to see directories like "tags" and "releases" next to "branches" and "trunk".

For example, the Twisted project has a subversion server on "svn.twistedmatrix.com" that hosts several sub-projects. The repository is available through a SCHEME of "svn:". The primary sub-project is Twisted, of course, with a repository root of "svn://svn.twistedmatrix.com/svn/Twisted". Another sub-project is Informant, with a root of "svn://svn.twistedmatrix.com/svn/Informant", etc. Inside any checked-out Twisted tree, there is a file named bin/trial (which is used to run unit test suites).

The trunk for Twisted is in "svn://svn.twistedmatrix.com/svn/Twisted/trunk", and the fully-qualified SVN URL for the trunk version of `trial` would be "svn://svn.twistedmatrix.com/svn/Twisted/trunk/bin/trial". The same SVNURL for that file on a branch named "1.5.x" would be "svn://svn.twistedmatrix.com/svn/Twisted/branches/1.5.x/bin/tr

To set up a `SVNPoller` that watches the Twisted trunk (and nothing else), we would use the following:

```
from buildbot.changes.svnpoller import SVNPoller
c['change_source'] = SVNPoller("svn://svn.twistedmatrix.com/svn/Twisted/trunk")
```

In this case, every Change that our `SVNPoller` produces will have `.branch=None`, to indicate that the Change is on the trunk. No other sub-projects or branches will be tracked.

If we want our ChangeSource to follow multiple branches, we have to do two things. First we have to change our `svnurl=` argument to watch more than just ".../Twisted/trunk". We will set it to ".../Twisted" so that we'll see both the trunk and all the branches. Second, we have to tell `SVNPoller` how to split the (PROJECT-plus-BRANCH)(FILEPATH) strings it gets from the repository out into (BRANCH) and (FILEPATH) pairs.

We do the latter by providing a "split_file" function. This function is responsible for splitting something like "branches/1.5.x/bin/trial" into `branch`="branches/1.5.x" and `filepath`="bin/trial". This function is always given a string that names a file relative to the subdirectory pointed to by the `SVNPoller`'s `svnurl=` argument. It is expected to return a (BRANCHNAME, FILEPATH) tuple (in which FILEPATH is relative to the branch indicated), or None to indicate that the file is outside any project of interest.

(note that we want to see "branches/1.5.x" rather than just "1.5.x" because when we perform the SVN checkout, we will probably append the branch name to the baseURL, which requires that we keep the "branches" component in there. Other VC schemes use a different approach towards branches and may not require this artifact.)

If your repository uses this same PROJECT/BRANCH/FILEPATH naming scheme, the following function will work:

```
def split_file_branches(path):
    pieces = path.split('/')
    if pieces[0] == 'trunk':
        return (None, '/'.join(pieces[1:]))
    elif pieces[0] == 'branches':
        return ('/'.join(pieces[0:2]),
                '/'.join(pieces[2:]))
    else:
        return None
```

This function is provided as `buildbot.changes.svnpoller.split_file_branches` for your convenience. So to have our Twisted-watching `SVNPoller` follow multiple branches, we would use this:

```
from buildbot.changes.svnpoller import SVNPoller, split_file_branches
c['change_source'] = SVNPoller("svn://svn.twistedmatrix.com/svn/Twisted",
                               split_file=split_file_branches)
```

Changes for all sorts of branches (with names like "branches/1.5.x", and None to indicate the trunk) will be delivered to the Schedulers. Each Scheduler is then free to use or ignore each branch as it sees fit.

BRANCHNAME/PROJECT/FILEPATH repositories

Another common way to organize a Subversion repository is to put the branch name at the top, and the projects underneath. This is especially frequent when there are a number of related sub-projects that all get released in a group.

For example, Divmod.org hosts a project named "Nevow" as well as one named "Quotient". In a checked-out Nevow tree there is a directory named "formless" that contains a python source file named "webform.py". This repository is accessible via webdav (and thus uses an "http:" scheme) through the divmod.org hostname. There are many branches in this repository, and they use a (BRANCHNAME)/(PROJECT) naming policy.

The fully-qualified SVN URL for the trunk version of webform.py is `http://divmod.org/svn/Divmod/trunk/Nevow/formless/webform.py`. You can do an `svn co` with that URL and get a copy of the latest version. The 1.5.x branch version of this file would have a URL of `http://divmod.org/svn/Divmod/branches/1.5.x/Nevow/formless/webform.p` The whole Nevow trunk would be checked out with `http://divmod.org/svn/Divmod/trunk/Nevow`, while the Quotient trunk would be checked out using `http://divmod.org/svn/Divmod/trunk/Quotient.`

Now suppose we want to have an `SVNPoller` that only cares about the Nevow trunk. This case looks just like the PROJECT/BRANCH layout described earlier:

```
from buildbot.changes.svnpoller import SVNPoller
c['change_source'] = SVNPoller("http://divmod.org/svn/Divmod/trunk/Nevow")
```

But what happens when we want to track multiple Nevow branches? We have to point our `svnurl=` high enough to see all those branches, but we also don't want to include Quotient changes (since we're only building Nevow). To accomplish this, we must rely upon the `split_file` function to help us tell the difference between files that belong to Nevow and those that belong to Quotient, as well as figuring out which branch each one is on.

```
from buildbot.changes.svnpoller import SVNPoller
c['change_source'] = SVNPoller("http://divmod.org/svn/Divmod",
                               split_file=my_file_splitter)
```

The `my_file_splitter` function will be called with repository-relative pathnames like:

`trunk/Nevow/formless/webform.py`
: This is a Nevow file, on the trunk. We want the Change that includes this to see a filename of `formless/webform.py`", and a branch of None

`branches/1.5.x/Nevow/formless/webform.py`
: This is a Nevow file, on a branch. We want to get branch="branches/1.5.x" and filename="formless/webform.py".

`trunk/Quotient/setup.py`
: This is a Quotient file, so we want to ignore it by having `my_file_splitter` return None.

`branches/1.5.x/Quotient/setup.py`
: This is also a Quotient file, which should be ignored.

The following definition for `my_file_splitter` will do the job:

```
def my_file_splitter(path):
    pieces = path.split('/')
    if pieces[0] == 'trunk':
        branch = None
        pieces.pop(0) # remove 'trunk'
    elif pieces[0] == 'branches':
        pieces.pop(0) # remove 'branches'
        # grab branch name
        branch = 'branches/' + pieces.pop(0)
    else:
        return None # something weird
    projectname = pieces.pop(0)
    if projectname != 'Nevow':
        return None # wrong project
    return (branch, '/'.join(pieces))
```

5.9 MercurialHook

Since Mercurial is written in python, the hook script can invoke Buildbot's `sendchange` function directly, rather than having to spawn an external process. This function delivers the same sort of changes as `buildbot sendchange` and the various hook scripts in contrib/, so you'll need to add a `pb.PBChangeSource` to your buildmaster to receive these changes.

To set this up, first choose a Mercurial repository that represents your central "official" source tree. This will be the same repository that your buildslaves will eventually pull from. Install Buildbot on the machine that hosts this repository, using the same version of python as Mercurial is using (so that the Mercurial hook can import code from buildbot). Then add the following to the `.hg/hgrc` file in that repository, replacing the buildmaster hostname/portnumber as appropriate for your buildbot:

```
[hooks]
changegroup.buildbot = python:buildbot.changes.hgbuildbot.hook

[hgbuildbot]
master = buildmaster.example.org:9987
```

(Note that Mercurial lets you define multiple `changegroup` hooks by giving them distinct names, like `changegroup.foo` and `changegroup.bar`, which is why we use `changegroup.buildbot` in this example. There is nothing magical about the "buildbot" suffix in the hook name. The `[hgbuildbot]` section *is* special, however, as it is the only section that the buildbot hook pays attention to.)

Also note that this runs as a `changegroup` hook, rather than as an `incoming` hook. The `changegroup` hook is run with multiple revisions at a time (say, if multiple revisions are being pushed to this repository in a single `hg push` command), whereas the `incoming` hook is run with just one revision at a time. The `hgbuildbot.hook` function will only work with the `changegroup` hook.

The `[hgbuildbot]` section has two other parameters that you might specify, both of which control the name of the branch that is attached to the changes coming from this hook.

One common branch naming policy for Mercurial repositories is to use it just like Darcs: each branch goes into a separate repository, and all the branches for a single project share a common parent directory. For example, you might have '`/var/repos/PROJECT/trunk/`' and '`/var/repos/PROJECT/release`'. To use this style, use the `branchtype = dirname` setting, which simply uses the last component of the repository's enclosing directory as the branch name:

```
[hgbuildbot]
master = buildmaster.example.org:9987
branchtype = dirname
```

Another approach is to use Mercurial's built-in branches (the kind created with `hg branch` and listed with `hg branches`). This feature associates persistent names with particular lines of descent within a single repository. (note that the buildbot `source.Mercurial` checkout step does not yet support this kind of branch). To have the commit hook deliver this sort of branch name with the Change object, use `branchtype = inrepo`:

```
[hgbuildbot]
master = buildmaster.example.org:9987
branchtype = inrepo
```

Finally, if you want to simply specify the branchname directly, for all changes, use `branch = BRANCHNAME`. This overrides `branchtype`:

```
[hgbuildbot]
master = buildmaster.example.org:9987
branch = trunk
```

If you use `branch=` like this, you'll need to put a separate .hgrc in each repository. If you use `branchtype=`, you may be able to use the same .hgrc for all your repositories, stored in '`~/.hgrc`' or '`/etc/mercurial/hgrc`'.

5.10 Bzr Hook

Bzr is also written in Python, and the Bzr hook depends on Twisted to send the changes.

To install, put `contrib/bzr_buildbot.py` in one of your plugins locations a bzr plugins directory (e.g., `~/.bazaar/plugins`). Then, in one of your bazaar conf files (e.g., `~/.bazaar/locations.conf`), set the location you want to connect with buildbot with these keys:

`buildbot_on`
: one of 'commit', 'push, or 'change'. Turns the plugin on to report changes via commit, changes via push, or any changes to the trunk. 'change' is recommended.

`buildbot_server`
: (required to send to a buildbot master) the URL of the buildbot master to which you will connect (as of this writing, the same server and port to which slaves connect).

`buildbot_port`
: (optional, defaults to 9989) the port of the buildbot master to which you will connect (as of this writing, the same server and port to which slaves connect)

`buildbot_pqm`
: (optional, defaults to not pqm) Normally, the user that commits the revision is the user that is responsible for the change. When run in a pqm (Patch Queue Manager, see https://launchpad.net/pqm) environment, the user that commits is the Patch Queue Manager, and the user that committed the *parent* revision is responsible for the change. To turn on the pqm mode, set this value to any of (case-insensitive) `"Yes"`, `"Y"`, `"True"`, or `"T"`.

`buildbot_dry_run`
: (optional, defaults to not a dry run) Normally, the post-commit hook will attempt to communicate with the configured buildbot server and port. If this parameter is included and any of (case-insensitive) `"Yes"`, `"Y"`, `"True"`, or `"T"`, then the hook will simply print what it would have sent, but not attempt to contact the buildbot master.

`buildbot_send_branch_name`
: (optional, defaults to not sending the branch name) If your buildbot's bzr source build step uses a repourl, do *not* turn this on. If your buildbot's bzr build step uses a baseURL, then you may set this value to any of (case-insensitive) `"Yes"`, `"Y"`, `"True"`, or `"T"` to have the buildbot master append the branch name to the baseURL.

When buildbot no longer has a hardcoded password, it will be a configuration option here as well.

Here's a simple example that you might have in your `~/.bazaar/locations.conf`.

```
[chroot-*:///var/local/myrepo/mybranch]
buildbot_on = change
buildbot_server = localhost
```

5.11 Bzr Poller

If you cannot insert a Bzr hook in the server, you can use the Bzr Poller. To use, put `contrib/bzr_buildbot.py` somewhere that your buildbot configuration can import it. Even putting it in the same directory as the master.cfg should work. Install the poller in the buildbot configuration as with any other change source. Minimally, provide a URL that you want to poll (bzr://, bzr+ssh://, or lp:), though make sure the buildbot user has necessary privileges. You may also want to specify these optional values.

`poll_interval`
: The number of seconds to wait between polls. Defaults to 10 minutes.

`branch_name`
: Any value to be used as the branch name. Defaults to None, or specify a string, or specify the constants from `bzr_buildbot.py` SHORT or FULL to get the short branch name or full branch address.

`blame_merge_author`
: normally, the user that commits the revision is the user that is responsible for the change. When run in a pqm (Patch Queue Manager, see https://launchpad.net/pqm) environment, the user that commits is the Patch Queue Manager, and the user that committed the merged, *parent* revision is responsible for the change. set this value to True if this is pointed against a PQM-managed branch.

6 Build Process

A `Build` object is responsible for actually performing a build. It gets access to a remote `SlaveBuilder` where it may run commands, and a `BuildStatus` object where it must emit status events. The `Build` is created by the Builder's `BuildFactory`.

The default `Build` class is made up of a fixed sequence of `BuildSteps`, executed one after another until all are complete (or one of them indicates that the build should be halted early). The default `BuildFactory` creates instances of this `Build` class with a list of `BuildSteps`, so the basic way to configure the build is to provide a list of `BuildSteps` to your `BuildFactory`.

More complicated `Build` subclasses can make other decisions: execute some steps only if certain files were changed, or if certain previous steps passed or failed. The base class has been written to allow users to express basic control flow without writing code, but you can always subclass and customize to achieve more specialized behavior.

6.1 Build Steps

`BuildStep`s are usually specified in the buildmaster's configuration file, in a list that goes into the `BuildFactory`. The `BuildStep` instances in this list are used as templates to construct new independent copies for each build (so that state can be kept on the `BuildStep` in one build without affecting a later build). Each `BuildFactory` can be created with a list of steps, or the factory can be created empty and then steps added to it using the `addStep` method:

```
from buildbot.steps import source, shell
from buildbot.process import factory

f = factory.BuildFactory()
f.addStep(source.SVN(svnurl="http://svn.example.org/Trunk/"))
f.addStep(shell.ShellCommand(command=["make", "all"]))
f.addStep(shell.ShellCommand(command=["make", "test"]))
```

In earlier versions (0.7.5 and older), these steps were specified with a tuple of (step_class, keyword_arguments). Steps can still be specified this way, but the preferred form is to pass actual `BuildStep` instances to `addStep`, because that gives the `BuildStep` class a chance to do some validation on the arguments.

If you have a common set of steps which are used in several factories, the `addSteps` method may be handy. It takes an iterable of `BuildStep` instances.

```
setup_steps = [
    source.SVN(svnurl="http://svn.example.org/Trunk/")
    shell.ShellCommand(command="./setup")
]
quick = factory.BuildFactory()
quick.addSteps(setup_steps)
quick.addStep(shell.shellCommand(command="make quick"))
```

The rest of this section lists all the standard BuildStep objects available for use in a Build, and the parameters which can be used to control each.

6.1.1 Common Parameters

The standard `Build` runs a series of `BuildSteps` in order, only stopping when it runs out of steps or if one of them requests that the build be halted. It collects status information from each one to create an overall build status (of SUCCESS, WARNINGS, or FAILURE).

All BuildSteps accept some common parameters. Some of these control how their individual status affects the overall build. Others are used to specify which `Locks` (see see Section 6.2 [Interlocks], page 96) should be acquired before allowing the step to run.

Arguments common to all `BuildStep` subclasses:

`name` the name used to describe the step on the status display. It is also used to give a name to any LogFiles created by this step.

`haltOnFailure`
if True, a FAILURE of this build step will cause the build to halt immediately. Steps with `alwaysRun=True` are still run. Generally speaking, haltOnFailure implies flunkOnFailure (the default for most BuildSteps). In some cases, particularly series of tests, it makes sense to haltOnFailure if something fails early on but not flunkOnFailure. This can be achieved with haltOnFailure=True, flunkOnFailure=False.

`flunkOnWarnings`
when True, a WARNINGS or FAILURE of this build step will mark the overall build as FAILURE. The remaining steps will still be executed.

`flunkOnFailure`
when True, a FAILURE of this build step will mark the overall build as a FAILURE. The remaining steps will still be executed.

`warnOnWarnings`
when True, a WARNINGS or FAILURE of this build step will mark the overall build as having WARNINGS. The remaining steps will still be executed.

`warnOnFailure`
when True, a FAILURE of this build step will mark the overall build as having WARNINGS. The remaining steps will still be executed.

`alwaysRun`
if True, this build step will always be run, even if a previous buildstep with `haltOnFailure=True` has failed.

`doStepIf` A step can be configured to only run under certain conditions. To do this, set the step's `doStepIf` to a boolean value, or to a function that returns a boolean value. If the value or function result is false, then the step will return SKIPPED without doing anything. Oherwise, the step will be executed normally. If you set `doStepIf` to a function, that function should accept one parameter, which will be the `Step` object itself.

`locks` a list of Locks (instances of `buildbot.locks.SlaveLock` or `buildbot.locks.MasterLock`) that should be acquired before starting this Step. The Locks will be released when the step is complete. Note that this is a list of actual Lock instances, not names. Also note that all Locks must have unique names.

6.1.2 Using Build Properties

Build properties are a generalized way to provide configuration information to build steps; see Section 3.7 [Build Properties], page 27.

Some build properties are inherited from external sources – global properties, schedulers, or buildslaves. Some build properties are set when the build starts, such as the SourceStamp information. Other properties can be set by BuildSteps as they run, for example the various Source steps will set the `got_revision` property to the source revision that was actually checked out (which can be useful when the SourceStamp in use merely requested the "latest revision": `got_revision` will tell you what was actually built).

In custom BuildSteps, you can get and set the build properties with the `getProperty`/`setProperty` methods. Each takes a string for the name of the property, and returns or accepts an arbitrary[1] object. For example:

```
class MakeTarball(ShellCommand):
    def start(self):
        if self.getProperty("os") == "win":
            self.setCommand([ ... ]) # windows-only command
        else:
            self.setCommand([ ... ]) # equivalent for other systems
        ShellCommand.start(self)
```

WithProperties

You can use build properties in ShellCommands by using the `WithProperties` wrapper when setting the arguments of the ShellCommand. This interpolates the named build properties into the generated shell command. Most step parameters accept `WithProperties`. Please file bugs for any parameters which do not.

```
from buildbot.steps.shell import ShellCommand
from buildbot.process.properties import WithProperties

f.addStep(ShellCommand(
          command=["tar", "czf",
                   WithProperties("build-%s.tar.gz", "revision"),
                   "source"]))
```

If this BuildStep were used in a tree obtained from Subversion, it would create a tarball with a name like '`build-1234.tar.gz`'.

The `WithProperties` function does `printf`-style string interpolation, using strings obtained by calling `build.getProperty(propname)`. Note that for every `%s` (or `%d`, etc), you must have exactly one additional argument to indicate which build property you want to insert.

You can also use python dictionary-style string interpolation by using the `%(propname)s` syntax. In this form, the property name goes in the parentheses, and WithProperties takes *no* additional arguments:

[1] Build properties are serialized along with the build results, so they must be serializable. For this reason, the value of any build property should be simple inert data: strings, numbers, lists, tuples, and dictionaries. They should not contain class instances.

```
f.addStep(ShellCommand(
          command=["tar", "czf",
                   WithProperties("build-%(revision)s.tar.gz"),
                   "source"]))
```

Don't forget the extra "s" after the closing parenthesis! This is the cause of many confusing errors.

The dictionary-style interpolation supports a number of more advanced syntaxes, too.

`propname:-replacement`
: If `propname` exists, substitute its value; otherwise, substitute `replacement`. `replacement` may be empty (`%(propname:-)s`)

`propname:+replacement`
: If `propname` exists, substitute `replacement`; otherwise, substitute an empty string.

Although these are similar to shell substitutions, no other substitutions are currently supported, and `replacement` in the above cannot contain more substitutions.

Note: like python, you can either do positional-argument interpolation *or* keyword-argument interpolation, not both. Thus you cannot use a string like `WithProperties("foo-%(revision)s-%s", "branch")`.

Common Build Properties

The following build properties are set when the build is started, and are available to all steps.

`branch`
: This comes from the build's SourceStamp, and describes which branch is being checked out. This will be `None` (which interpolates into `WithProperties` as an empty string) if the build is on the default branch, which is generally the trunk. Otherwise it will be a string like "branches/beta1.4". The exact syntax depends upon the VC system being used.

`revision`
: This also comes from the SourceStamp, and is the revision of the source code tree that was requested from the VC system. When a build is requested of a specific revision (as is generally the case when the build is triggered by Changes), this will contain the revision specification. This is always a string, although the syntax depends upon the VC system in use: for SVN it is an integer, for Mercurial and Monotone it is a short string, for Darcs it is a rather large string, etc.

  If the "force build" button was pressed, the revision will be `None`, which means to use the most recent revision available. This is a "trunk build". This will be interpolated as an empty string.

`got_revision`
: This is set when a Source step checks out the source tree, and provides the revision that was actually obtained from the VC system. In general this should be

the same as `revision`, except for trunk builds, where `got_revision` indicates what revision was current when the checkout was performed. This can be used to rebuild the same source code later.

Note that for some VC systems (Darcs in particular), the revision is a large string containing newlines, and is not suitable for interpolation into a filename.

`buildername`
: This is a string that indicates which Builder the build was a part of. The combination of buildername and buildnumber uniquely identify a build.

`buildnumber`
: Each build gets a number, scoped to the Builder (so the first build performed on any given Builder will have a build number of 0). This integer property contains the build's number.

`slavename`
: This is a string which identifies which buildslave the build is running on.

`scheduler`
: If the build was started from a scheduler, then this property will contain the name of that scheduler.

6.1.3 Source Checkout

The first step of any build is typically to acquire the source code from which the build will be performed. There are several classes to handle this, one for each of the different source control system that Buildbot knows about. For a description of how Buildbot treats source control in general, see Section 3.1 [Version Control Systems], page 17.

All source checkout steps accept some common parameters to control how they get the sources and where they should be placed. The remaining per-VC-system parameters are mostly to specify where exactly the sources are coming from.

`mode`
: a string describing the kind of VC operation that is desired. Defaults to `update`.

  `update` specifies that the CVS checkout/update should be performed directly into the workdir. Each build is performed in the same directory, allowing for incremental builds. This minimizes disk space, bandwidth, and CPU time. However, it may encounter problems if the build process does not handle dependencies properly (sometimes you must do a "clean build" to make sure everything gets compiled), or if source files are deleted but generated files can influence test behavior (e.g. python's .pyc files), or when source directories are deleted but generated files prevent CVS from removing them. Builds ought to be correct regardless of whether they are done "from scratch" or incrementally, but it is useful to test both kinds: this mode exercises the incremental-build style.

  `copy` specifies that the CVS workspace should be maintained in a separate directory (called the 'copydir'), using checkout or update as necessary. For each build, a new workdir is created with a copy of

the source tree (rm -rf workdir; cp -r copydir workdir). This doubles the disk space required, but keeps the bandwidth low (update instead of a full checkout). A full 'clean' build is performed each time. This avoids any generated-file build problems, but is still occasionally vulnerable to CVS problems such as a repository being manually rearranged, causing CVS errors on update which are not an issue with a full checkout.

`clobber` specifes that the working directory should be deleted each time, necessitating a full checkout for each build. This insures a clean build off a complete checkout, avoiding any of the problems described above. This mode exercises the "from-scratch" build style.

`export` this is like `clobber`, except that the 'cvs export' command is used to create the working directory. This command removes all CVS metadata files (the CVS/ directories) from the tree, which is sometimes useful for creating source tarballs (to avoid including the metadata in the tar file).

`workdir` like all Steps, this indicates the directory where the build will take place. Source Steps are special in that they perform some operations outside of the workdir (like creating the workdir itself).

`alwaysUseLatest`
if True, bypass the usual "update to the last Change" behavior, and always update to the latest changes instead.

`retry` If set, this specifies a tuple of `(delay, repeats)` which means that when a full VC checkout fails, it should be retried up to *repeats* times, waiting *delay* seconds between attempts. If you don't provide this, it defaults to `None`, which means VC operations should not be retried. This is provided to make life easier for buildslaves which are stuck behind poor network connections.

My habit as a developer is to do a `cvs update` and `make` each morning. Problems can occur, either because of bad code being checked in, or by incomplete dependencies causing a partial rebuild to fail where a complete from-scratch build might succeed. A quick Builder which emulates this incremental-build behavior would use the `mode='update'` setting.

On the other hand, other kinds of dependency problems can cause a clean build to fail where a partial build might succeed. This frequently results from a link step that depends upon an object file that was removed from a later version of the tree: in the partial tree, the object file is still around (even though the Makefiles no longer know how to create it).

"official" builds (traceable builds performed from a known set of source revisions) are always done as clean builds, to make sure it is not influenced by any uncontrolled factors (like leftover files from a previous build). A "full" Builder which behaves this way would want to use the `mode='clobber'` setting.

Each VC system has a corresponding source checkout class: their arguments are described on the following pages.

6.1.3.1 CVS

The `CVS` build step performs a CVS (`http://www.nongnu.org/cvs/`) checkout or update. It takes the following arguments:

`cvsroot` (required): specify the CVSROOT value, which points to a CVS repository, probably on a remote machine. For example, the cvsroot value you would use to get a copy of the Buildbot source code is `:pserver:anonymous@cvs.sourceforge.net:/cvsroot/buildbot`

`cvsmodule`
(required): specify the cvs `module`, which is generally a subdirectory of the CVSROOT. The cvsmodule for the Buildbot source code is `buildbot`.

`branch` a string which will be used in a `-r` argument. This is most useful for specifying a branch to work on. Defaults to `HEAD`.

`global_options`
a list of flags to be put before the verb in the CVS command.

`checkoutDelay`
if set, the number of seconds to put between the timestamp of the last known Change and the value used for the `-D` option. Defaults to half of the parent Build's treeStableTimer.

6.1.3.2 SVN

The `SVN` build step performs a Subversion (`http://subversion.tigris.org`) checkout or update. There are two basic ways of setting up the checkout step, depending upon whether you are using multiple branches or not.

If all of your builds use the same branch, then you should create the `SVN` step with the `svnurl` argument:

`svnurl` (required): this specifies the `URL` argument that will be given to the `svn checkout` command. It dictates both where the repository is located and which sub-tree should be extracted. In this respect, it is like a combination of the CVS `cvsroot` and `cvsmodule` arguments. For example, if you are using a remote Subversion repository which is accessible through HTTP at a URL of `http://svn.example.com/repos`, and you wanted to check out the `trunk/calc` sub-tree, you would use `svnurl="http://svn.example.com/repos/trunk/calc"` as an argument to your `SVN` step.

If, on the other hand, you are building from multiple branches, then you should create the `SVN` step with the `baseURL` and `defaultBranch` arguments instead:

`baseURL` (required): this specifies the base repository URL, to which a branch name will be appended. It should probably end in a slash.

`defaultBranch`
(optional): this specifies the name of the branch to use when a Build does not provide one of its own. This will be appended to `baseURL` to create the string that will be passed to the `svn checkout` command.

`username` (optional): if specified, this will be passed to the `svn` binary with a `--username` option.

`password` (optional): if specified, this will be passed to the `svn` binary with a `--password` option. The password itself will be suitably obfuscated in the logs.

`extra_args`
(optional): if specified, an array of strings that will be passed as extra arguments to the `svn` binary.

`keep_on_purge`
(optional): specific files or directories to keep between purges, like some build outputs that can be reused between builds.

`ignore_ignores`
(optional): when purging changes, don't use rules defined in svn:ignore properties and global-ignores in subversion/config.

`always_purge`
(optional): if set to True, always purge local changes after each build. This is everything that would appear in a `svn status`.

`depth` (optional): Specify depth argument to achieve sparse checkout. Only available if slave has Subversion 1.5 or higher.

If set to `"empty"` updates will not pull in any files or subdirectories not already present. If set to `"files"`, updates will pull in any files not already present, but not directories. If set to `"immediates"`, updates willl pull in any files or subdirectories not already present, the new subdirectories will have depth: empty. If set to `"infinity"`, updates will pull in any files or subdirectories not already present; the new subdirectories will have depth-infinity. Infinity is equivalent to SVN default update behavior, without specifying any depth argument.

If you are using branches, you must also make sure your `ChangeSource` will report the correct branch names.

branch example

Let's suppose that the "MyProject" repository uses branches for the trunk, for various users' individual development efforts, and for several new features that will require some amount of work (involving multiple developers) before they are ready to merge onto the trunk. Such a repository might be organized as follows:

```
svn://svn.example.org/MyProject/trunk
svn://svn.example.org/MyProject/branches/User1/foo
svn://svn.example.org/MyProject/branches/User1/bar
svn://svn.example.org/MyProject/branches/User2/baz
svn://svn.example.org/MyProject/features/newthing
svn://svn.example.org/MyProject/features/otherthing
```

Further assume that we want the Buildbot to run tests against the trunk and against all the feature branches (i.e., do a checkout/compile/build of branch X when a file has been changed on branch X, when X is in the set [trunk, features/newthing, features/otherthing]). We do not want the Buildbot to automatically build any of the user branches, but it should

be willing to build a user branch when explicitly requested (most likely by the user who owns that branch).

There are three things that need to be set up to accomodate this system. The first is a ChangeSource that is capable of identifying the branch which owns any given file. This depends upon a user-supplied function, in an external program that runs in the SVN commit hook and connects to the buildmaster's `PBChangeSource` over a TCP connection. (you can use the "`buildbot sendchange`" utility for this purpose, but you will still need an external program to decide what value should be passed to the `--branch=` argument). For example, a change to a file with the SVN url of "svn://svn.example.org/MyProject/features/newthing/src/foo.c" should be broken down into a Change instance with `branch='features/newthing'` and `file='src/foo.c'`.

The second piece is an `AnyBranchScheduler` which will pay attention to the desired branches. It will not pay attention to the user branches, so it will not automatically start builds in response to changes there. The AnyBranchScheduler class requires you to explicitly list all the branches you want it to use, but it would not be difficult to write a subclass which used `branch.startswith('features/'` to remove the need for this explicit list. Or, if you want to build user branches too, you can use AnyBranchScheduler with `branches=None` to indicate that you want it to pay attention to all branches.

The third piece is an `SVN` checkout step that is configured to handle the branches correctly, with a `baseURL` value that matches the way the ChangeSource splits each file's URL into base, branch, and file.

```
from buildbot.changes.pb import PBChangeSource
from buildbot.scheduler import AnyBranchScheduler
from buildbot.process import source, factory
from buildbot.steps import source, shell

c['change_source'] = PBChangeSource()
s1 = AnyBranchScheduler('main',
                        ['trunk', 'features/newthing', 'features/otherthing'],
                        10*60, ['test-i386', 'test-ppc'])
c['schedulers'] = [s1]

f = factory.BuildFactory()
f.addStep(source.SVN(mode='update',
                     baseURL='svn://svn.example.org/MyProject/',
                     defaultBranch='trunk'))
f.addStep(shell.Compile(command="make all"))
f.addStep(shell.Test(command="make test"))

c['builders'] = [
  {'name':'test-i386', 'slavename':'bot-i386', 'builddir':'test-i386',
                       'factory':f },
  {'name':'test-ppc', 'slavename':'bot-ppc', 'builddir':'test-ppc',
                      'factory':f },
 ]
```

In this example, when a change arrives with a `branch` attribute of "trunk", the resulting build will have an SVN step that concatenates "svn://svn.example.org/MyProject/" (the baseURL) with "trunk" (the branch name) to get the correct svn command. If the "newthing" branch has a change to "src/foo.c", then the SVN step will concatenate "svn://svn.example.org/MyProject/" with "features/newthing" to get the svnurl for checkout.

6.1.3.3 Darcs

The `Darcs` build step performs a Darcs (`http://darcs.net/`) checkout or update.

Like See Section 6.1.3.2 [SVN], page 72, this step can either be configured to always check out a specific tree, or set up to pull from a particular branch that gets specified separately for each build. Also like SVN, the repository URL given to Darcs is created by concatenating a `baseURL` with the branch name, and if no particular branch is requested, it uses a `defaultBranch`. The only difference in usage is that each potential Darcs repository URL must point to a fully-fledged repository, whereas SVN URLs usually point to sub-trees of the main Subversion repository. In other words, doing an SVN checkout of `baseURL` is legal, but silly, since you'd probably wind up with a copy of every single branch in the whole repository. Doing a Darcs checkout of `baseURL` is just plain wrong, since the parent directory of a collection of Darcs repositories is not itself a valid repository.

The Darcs step takes the following arguments:

`repourl` (required unless `baseURL` is provided): the URL at which the Darcs source repository is available.

`baseURL` (required unless `repourl` is provided): the base repository URL, to which a branch name will be appended. It should probably end in a slash.

`defaultBranch`
(allowed if and only if `baseURL` is provided): this specifies the name of the branch to use when a Build does not provide one of its own. This will be appended to `baseURL` to create the string that will be passed to the `darcs get` command.

6.1.3.4 Mercurial

The `Mercurial` build step performs a Mercurial (`http://selenic.com/mercurial`) (aka "hg") checkout or update.

Branches are available in two modes: "dirname" like See Section 6.1.3.3 [Darcs], page 75, or "inrepo", which uses the repository internal branches. Make sure this setting matches your changehook, if you have that installed.

The Mercurial step takes the following arguments:

`repourl` (required unless `baseURL` is provided): the URL at which the Mercurial source repository is available.

`baseURL` (required unless `repourl` is provided): the base repository URL, to which a branch name will be appended. It should probably end in a slash.

`defaultBranch`
(allowed if and only if `baseURL` is provided): this specifies the name of the branch to use when a Build does not provide one of its own. This will be

appended to `baseURL` to create the string that will be passed to the `hg clone` command.

`branchType`
: either 'dirname' (default) or 'inrepo' depending on whether the branch name should be appended to the `baseURL` or the branch is a mercurial named branch and can be found within the `repourl`.

`clobberOnBranchChange`
: boolean, defaults to True. If set and using inrepos branches, clobber the tree at each branch change. Otherwise, just update to the branch.

6.1.3.5 Arch

The `Arch` build step performs an Arch (`http://gnuarch.org/`) checkout or update using the `tla` client. It takes the following arguments:

`url` (required): this specifies the URL at which the Arch source archive is available.

`version` (required): this specifies which "development line" (like a branch) should be used. This provides the default branch name, but individual builds may specify a different one.

`archive` (optional): Each repository knows its own archive name. If this parameter is provided, it must match the repository's archive name. The parameter is accepted for compatibility with the `Bazaar` step, below.

6.1.3.6 Bazaar

`Bazaar` is an alternate implementation of the Arch VC system, which uses a client named `baz`. The checkout semantics are just different enough from `tla` that there is a separate BuildStep for it.

It takes exactly the same arguments as `Arch`, except that the `archive=` parameter is required. (baz does not emit the archive name when you do `baz register-archive`, so we must provide it ourselves).

6.1.3.7 Bzr

`bzr` is a descendant of Arch/Baz, and is frequently referred to as simply "Bazaar". The repository-vs-workspace model is similar to Darcs, but it uses a strictly linear sequence of revisions (one history per branch) like Arch. Branches are put in subdirectories. This makes it look very much like Mercurial. It takes the following arguments:

`repourl` (required unless `baseURL` is provided): the URL at which the Bzr source repository is available.

`baseURL` (required unless `repourl` is provided): the base repository URL, to which a branch name will be appended. It should probably end in a slash.

`defaultBranch`
: (allowed if and only if `baseURL` is provided): this specifies the name of the branch to use when a Build does not provide one of its own. This will be appended to `baseURL` to create the string that will be passed to the `bzr checkout` command.

`forceSharedRepo`
: (boolean, optional, defaults to False): If set to True, the working directory will be made into a bzr shared repository if it is not already. Shared repository greatly reduces the amount of history data that needs to be downloaded if not using update/copy mode, or if using update/copy mode with multiple branches.

6.1.3.8 P4

The `P4` build step creates a Perforce (`http://www.perforce.com/`) client specification and performs an update.

`p4base`
: A view into the Perforce depot without branch name or trailing `"..."`. Typically `"//depot/proj/"`.

`defaultBranch`
: A branch name to append on build requests if none is specified. Typically `"trunk"`.

`p4port`
: (optional): the host:port string describing how to get to the P4 Depot (repository), used as the -p argument for all p4 commands.

`p4user`
: (optional): the Perforce user, used as the -u argument to all p4 commands.

`p4passwd`
: (optional): the Perforce password, used as the -p argument to all p4 commands.

`p4extra_views`
: (optional): a list of (depotpath, clientpath) tuples containing extra views to be mapped into the client specification. Both will have `"/..."` appended automatically. The client name and source directory will be prepended to the client path.

`p4client`
: (optional): The name of the client to use. In mode='copy' and mode='update', it's particularly important that a unique name is used for each checkout directory to avoid incorrect synchronization. For this reason, Python percent substitution will be performed on this value to replace %(slave)s with the slave name and %(builder)s with the builder name. The default is `"buildbot_%(slave)s_%(build)s"`.

6.1.3.9 Git

The `Git` build step clones or updates a Git (`http://git.or.cz/`) repository and checks out the specified branch or revision. Note that the buildbot supports Git version 1.2.0 and later: earlier versions (such as the one shipped in Ubuntu 'Dapper') do not support the `git init` command that the buildbot uses.

The Git step takes the following arguments:

`repourl`
: (required): the URL of the upstream Git repository.

`branch`
: (optional): this specifies the name of the branch to use when a Build does not provide one of its own. If this this parameter is not specified, and the Build does not provide a branch, the "master" branch will be used.

`ignore_ignores`
: (optional): when purging changes, don't use .gitignore and .git/info/exclude.

6.1.3.10 Monotone

The `Monotone` build step performs a Monotone (`http://code.monotone.ca/monotone`) (aka "mtn") checkout or update.

It takes the following arguments:

`repourl` The server address and port (addr:port) or URI where the Monotone source repositry is available. This is a required argument.

`branch` The default branch that will be used when a Build does not provide one of its own. This is a required argument.

6.1.4 ShellCommand

This is a useful base class for just about everything you might want to do during a build (except for the initial source checkout). It runs a single command in a child shell on the buildslave. All stdout/stderr is recorded into a LogFile. The step finishes with a status of FAILURE if the command's exit code is non-zero, otherwise it has a status of SUCCESS.

The preferred way to specify the command is with a list of argv strings, since this allows for spaces in filenames and avoids doing any fragile shell-escaping. You can also specify the command with a single string, in which case the string is given to '/bin/sh -c COMMAND' for parsing.

On Windows, commands are run via `cmd.exe /c` which works well. However, if you're running a batch file, the error level does not get propagated correctly unless you add 'call' before your batch file's name: `cmd=['call', 'myfile.bat', ...]`.

`ShellCommand` arguments:

`command` a list of strings (preferred) or single string (discouraged) which specifies the command to be run. A list of strings is preferred because it can be used directly as an argv array. Using a single string (with embedded spaces) requires the buildslave to pass the string to /bin/sh for interpretation, which raises all sorts of difficult questions about how to escape or interpret shell metacharacters.

`workdir` All ShellCommands are run by default in the "workdir", which defaults to the "`build`" subdirectory of the slave builder's base directory. The absolute path of the workdir will thus be the slave's basedir (set as an option to `buildbot create-slave`, see Section 2.5 [Creating a buildslave], page 10) plus the builder's basedir (set in the builder's `c['builddir']` key in master.cfg) plus the workdir itself (a class-level attribute of the BuildFactory, defaults to "`build`").

For example:

```
f.addStep(ShellCommand(command=["make", "test"],
                       workdir="build/tests"))
```

`env` a dictionary of environment strings which will be added to the child command's environment. For example, to run tests with a different i18n language setting, you might use

```
f.addStep(ShellCommand(command=["make", "test"],
                       env={'LANG': 'fr_FR'}))
```

These variable settings will override any existing ones in the buildslave's environment or the environment specified in the Builder. The exception is PYTHONPATH, which is merged with (actually prepended to) any existing $PYTHONPATH setting. The value is treated as a list of directories to prepend, and a single string is treated like a one-item list. For example, to prepend both '`/usr/local/lib/python2.3`' and '`/home/buildbot/lib/python`' to any existing $PYTHONPATH setting, you would do something like the following:

```
f.addStep(ShellCommand(
              command=["make", "test"],
              env={'PYTHONPATH': ["/usr/local/lib/python2.3",
                                  "/home/buildbot/lib/python"] }))
```

`want_stdout`
: if False, stdout from the child process is discarded rather than being sent to the buildmaster for inclusion in the step's LogFile.

`want_stderr`
: like `want_stdout` but for stderr. Note that commands run through a PTY do not have separate stdout/stderr streams: both are merged into stdout.

`usePTY`
: Should this command be run in a `pty`? The default is to observe the configuration of the client (see Section 2.5.1 [Buildslave Options], page 12), but specifying `True` or `False` here will override the default.

  The advantage of using a PTY is that "grandchild" processes are more likely to be cleaned up if the build is interrupted or times out (since it enables the use of a "process group" in which all child processes will be placed). The disadvantages: some forms of Unix have problems with PTYs, some of your unit tests may behave differently when run under a PTY (generally those which check to see if they are being run interactively), and PTYs will merge the stdout and stderr streams into a single output stream (which means the red-vs-black coloring in the logfiles will be lost).

`logfiles`
: Sometimes commands will log interesting data to a local file, rather than emitting everything to stdout or stderr. For example, Twisted's "trial" command (which runs unit tests) only presents summary information to stdout, and puts the rest into a file named '`_trial_temp/test.log`'. It is often useful to watch these files as the command runs, rather than using `/bin/cat` to dump their contents afterwards.

  The `logfiles=` argument allows you to collect data from these secondary logfiles in near-real-time, as the step is running. It accepts a dictionary which maps from a local Log name (which is how the log data is presented in the build results) to either a remote filename (interpreted relative to the build's working directory), or a dictionary of options. Each named file will be polled on a regular basis (every couple of seconds) as the build runs, and any new text will be sent over to the buildmaster.

  If you provide a dictionary of options instead of a string, you must specify the `filename` key. You can optionally provide a `follow` key which is a boolean

controlling whether a logfile is followed or concatenated in its entirety. Following is appropriate for logfiles to which the build step will append, where the pre-existing contents are not interesting. The default value for `follow` is `False`, which gives the same behavior as just providing a string filename.

```
f.addStep(ShellCommand(
              command=["make", "test"],
              logfiles={"triallog": "_trial_temp/test.log"}))
f.addStep(ShellCommand(
              command=["make", "test"],
              logfiles={"triallog": {"filename": "_trial_temp/test.log",
        "follow": True,}}))
```

`lazylogfiles`
: If set to `True`, logfiles will be tracked lazily, meaning that they will only be added when and if something is written to them. This can be used to suppress the display of empty or missing log files. The default is `False`.

`timeout` if the command fails to produce any output for this many seconds, it is assumed to be locked up and will be killed.

`maxTime` if the command takes longer than this many seconds, it will be killed.

`description`
: This will be used to describe the command (on the Waterfall display) while the command is still running. It should be a single imperfect-tense verb, like "compiling" or "testing". The preferred form is a list of short strings, which allows the HTML Waterfall display to create narrower columns by emitting a
 tag between each word. You may also provide a single string.

`descriptionDone`
: This will be used to describe the command once it has finished. A simple noun like "compile" or "tests" should be used. Like `description`, this may either be a list of short strings or a single string.

  If neither `description` nor `descriptionDone` are set, the actual command arguments will be used to construct the description. This may be a bit too wide to fit comfortably on the Waterfall display.

```
f.addStep(ShellCommand(command=["make", "test"],
                       description=["testing"],
                       descriptionDone=["tests"]))
```

`logEnviron`
: If this option is true (the default), then the step's logfile will describe the environment variables on the slave. In situations where the environment is not relevant and is long, it may be easier to set `logEnviron=False`.

6.1.5 Simple ShellCommand Subclasses

Several subclasses of ShellCommand are provided as starting points for common build steps. These are all very simple: they just override a few parameters so you don't have to specify them yourself, making the master.cfg file less verbose.

6.1.5.1 Configure

This is intended to handle the `./configure` step from autoconf-style projects, or the `perl Makefile.PL` step from perl MakeMaker.pm-style modules. The default command is `./configure` but you can change this by providing a `command=` parameter.

6.1.5.2 Compile

This is meant to handle compiling or building a project written in C. The default command is `make all`. When the compile is finished, the log file is scanned for GCC warning messages, a summary log is created with any problems that were seen, and the step is marked as WARNINGS if any were discovered. The number of warnings is stored in a Build Property named "warnings-count", which is accumulated over all Compile steps (so if two warnings are found in one step, and three are found in another step, the overall build will have a "warnings-count" property of 5.

The default regular expression used to detect a warning is `'.*warning[:].*'` , which is fairly liberal and may cause false-positives. To use a different regexp, provide a `warningPattern=` argument, or use a subclass which sets the `warningPattern` attribute:

```
f.addStep(Compile(command=["make", "test"],
                  warningPattern="^Warning: "))
```

The `warningPattern=` can also be a pre-compiled python regexp object: this makes it possible to add flags like `re.I` (to use case-insensitive matching).

The `suppressionFile=` argument can be specified as the (relative) path of a file inside the workdir defining warnings to be suppressed from the warning counting and log file. The file will be uploaded to the master from the slave before compiling, and any warning matched by a line in the suppression file will be ignored. This is useful to accept certain warnings (eg. in some special module of the source tree or in cases where the compiler is being particularly stupid), yet still be able to easily detect and fix the introduction of new warnings.

The file must contain one line per pattern of warnings to ignore. Empty lines and lines beginning with `#` are ignored. Other lines must consist of a regexp matching the file name, followed by a colon (`:`), followed by a regexp matching the text of the warning. Optionally this may be followed by another colon and a line number range. For example:

```
# Sample warning suppression file

mi_packrec.c : .*result of 32-bit shift implicitly converted to 64 bits.* : 560-600
DictTabInfo.cpp : .*invalid access to non-static.*
kernel_types.h : .*only defines private constructors and has no friends.* : 51
```

If no line number range is specified, the pattern matches the whole file; if only one number is given it matches only on that line.

The default warningPattern regexp only matches the warning text, so line numbers and file names are ignored. To enable line number and file name matching, privide a different regexp and provide a function (callable) as the argument of `warningExtractor=`. The function is called with three arguments: the BuildStep object, the line in the log file with the warning, and the `SRE_Match` object of the regexp search for `warningPattern`. It should return a tuple `(filename, linenumber, warning_test)`. For example:

```
f.addStep(Compile(command=["make"],
                  warningPattern="^(.*?):([0-9]+): [Ww]arning: (.*)$",
                  warningExtractor=Compile.warnExtractFromRegexpGroups,
                  suppressionFile="support-files/compiler_warnings.supp"))
```

(`Compile.warnExtractFromRegexpGroups` is a pre-defined function that returns the filename, linenumber, and text from groups (1,2,3) of the regexp match).

In projects with source files in multiple directories, it is possible to get full path names for file names matched in the suppression file, as long as the build command outputs the names of directories as they are entered into and left again. For this, specify regexps for the arguments `directoryEnterPattern=` and `directoryLeavePattern=`. The `directoryEnterPattern=` regexp should return the name of the directory entered into in the first matched group. The defaults, which are suitable for GNU Make, are these:

```
directoryEnterPattern = "make.*: Entering directory [\"‘’](.*)[’‘\"]"
directoryLeavePattern = "make.*: Leaving directory"
```

(TODO: this step needs to be extended to look for GCC error messages as well, and collect them into a separate logfile, along with the source code filenames involved).

6.1.5.3 Test

This is meant to handle unit tests. The default command is `make test`, and the `warnOnFailure` flag is set.

6.1.5.4 TreeSize

This is a simple command that uses the 'du' tool to measure the size of the code tree. It puts the size (as a count of 1024-byte blocks, aka 'KiB' or 'kibibytes') on the step's status text, and sets a build property named 'tree-size-KiB' with the same value.

6.1.5.5 PerlModuleTest

This is a simple command that knows how to run tests of perl modules. It parses the output to determine the number of tests passed and failed and total number executed, saving the results for later query.

6.1.6 Testing with mysql-test-run

The `process.mtrlogobserver.MTR` class is a subclass of `Test` (Section 6.1.5.3 [Test], page 82). It is used to run test suites using the mysql-test-run program, as used in MySQL, Drizzle, MariaDB, and MySQL storage engine plugins.

The shell command to run the test suite is specified in the same way as for the Test class. The MTR class will parse the output of running the test suite, and use the count of tests executed so far to provide more accurate completion time estimates. Any test failures that occur during the test are summarized on the Waterfall Display.

Server error logs are added as additional log files, useful to debug test failures.

Optionally, data about the test run and any test failures can be inserted into a database for further analysis and report generation. To use this facility, create an instance of `twisted.enterprise.adbapi.ConnectionPool` with connections to the database. The necessary tables can be created automatically by setting `autoCreateTables` to `True`, or manually using the SQL found in the '`mtrlogobserver.py`' source file.

One problem with specifying a database is that each reload of the configuration will get a new instance of `ConnectionPool` (even if the connection parameters are the same). To avoid that Buildbot thinks the builder configuration has changed because of this, use the `process.mtrlogobserver.EqConnectionPool` subclass of `ConnectionPool`, which implements an equiality operation that avoids this problem.

Example use:

```
from buildbot.process.mtrlogobserver import MTR, EqConnectionPool
myPool = EqConnectionPool("MySQLdb", "host", "buildbot", "password", "db")
myFactory.addStep(MTR(workdir="mysql-test", dbpool=myPool,
                      command=["perl", "mysql-test-run.pl", "--force"]))
```

`MTR` arguments:

`textLimit`
: Maximum number of test failures to show on the waterfall page (to not flood the page in case of a large number of test failures. Defaults to 5.

`testNameLimit`
: Maximum length of test names to show unabbreviated in the waterfall page, to avoid excessive column width. Defaults to 16.

`parallel`
: Value of `--parallel` option used for mysql-test-run.pl (number of processes used to run the test suite in parallel). Defaults to 4. This is used to determine the number of server error log files to download from the slave. Specifying a too high value does not hurt (as nonexisting error logs will be ignored), however if using `--parallel` value greater than the default it needs to be specified, or some server error logs will be missing.

`dbpool`
: An instance of twisted.enterprise.adbapi.ConnectionPool, or None. Defaults to None. If specified, results are inserted into the database using the ConnectionPool.

`autoCreateTables`
: Boolean, defaults to False. If True (and `dbpool` is specified), the necessary database tables will be created automatically if they do not exist already. Alternatively, the tables can be created manually from the SQL statements found in the mtrlogobserver.py source file.

`test_type`
: Short string that will be inserted into the database in the row for the test run. Defaults to the empty string, but can be specified to identify different types of test runs.

`test_info`
: Descriptive string that will be inserted into the database in the row for the test run. Defaults to the empty string, but can be specified as a user-readable description of this particular test run.

`mtr_subdir`
: The subdirectory in which to look for server error log files. Defaults to "mysql-test", which is usually correct. WithProperties is supported.

6.1.6.1 SetProperty

This buildstep is similar to ShellCommand, except that it captures the output of the command into a property. It is usually used like this:

```
f.addStep(SetProperty(command="uname -a", property="uname"))
```

This runs `uname -a` and captures its stdout, stripped of leading and trailing whitespace, in the property "uname". To avoid stripping, add `strip=False`. The `property` argument can be specified as a `WithProperties` object.

The more advanced usage allows you to specify a function to extract properties from the command output. Here you can use regular expressions, string interpolation, or whatever you would like. The function is called with three arguments: the exit status of the command, its standard output as a string, and its standard error as a string. It should return a dictionary containing all new properties.

```
def glob2list(rc, stdout, stderr):
    jpgs = [ l.strip() for l in stdout.split('\n') ]
    return { 'jpgs' : jpgs }
f.addStep(SetProperty(command="ls -1 *.jpg", extract_fn=glob2list))
```

Note that any ordering relationship of the contents of stdout and stderr is lost. For example, given

```
f.addStep(SetProperty(
    command="echo output1; echo error >&2; echo output2",
    extract_fn=my_extract))
```

Then `my_extract` will see `stdout="output1\noutput2\n"` and `stderr="error\n"`.

6.1.6.2 SubunitShellCommand

This buildstep is similar to ShellCommand, except that it runs the log content through a subunit filter to extract test and failure counts.

```
f.addStep(SubunitShellCommand(command="make test"))
```

This runs `make test` and filters it through subunit. The 'tests' and 'test failed' progress metrics will now accumulate test data from the test run.

6.1.7 Python BuildSteps

Here are some BuildSteps that are specifcally useful for projects implemented in Python.

6.1.7.1 BuildEPYDoc

epydoc (`http://epydoc.sourceforge.net/`) is a tool for generating API documentation for Python modules from their docstrings. It reads all the .py files from your source tree, processes the docstrings therein, and creates a large tree of .html files (or a single .pdf file).

The `buildbot.steps.python.BuildEPYDoc` step will run `epydoc` to produce this API documentation, and will count the errors and warnings from its output.

You must supply the command line to be used. The default is `make epydocs`, which assumes that your project has a Makefile with an "epydocs" target. You might wish to use something like `epydoc -o apiref source/PKGNAME` instead. You might also want to add `--pdf` to generate a PDF file instead of a large tree of HTML files.

The API docs are generated in-place in the build tree (under the workdir, in the subdirectory controlled by the "-o" argument). To make them useful, you will probably have to copy them to somewhere they can be read. A command like `rsync -ad apiref/ dev.example.com:~public_html/current-apiref/` might be useful. You might instead want to bundle them into a tarball and publish it in the same place where the generated install tarball is placed.

```
from buildbot.steps.python import BuildEPYDoc

...
f.addStep(BuildEPYDoc(command=["epydoc", "-o", "apiref", "source/mypkg"]))
```

6.1.7.2 PyFlakes

PyFlakes (`http://divmod.org/trac/wiki/DivmodPyflakes`) is a tool to perform basic static analysis of Python code to look for simple errors, like missing imports and references of undefined names. It is like a fast and simple form of the C "lint" program. Other tools (like pychecker) provide more detailed results but take longer to run.

The `buildbot.steps.python.PyFlakes` step will run pyflakes and count the various kinds of errors and warnings it detects.

You must supply the command line to be used. The default is `make pyflakes`, which assumes you have a top-level Makefile with a "pyflakes" target. You might want to use something like `pyflakes .` or `pyflakes src`.

```
from buildbot.steps.python import PyFlakes

...
f.addStep(PyFlakes(command=["pyflakes", "src"]))
```

6.1.7.3 PyLint

Similarly, the `buildbot.steps.python.PyLint` step will run pylint and analyze the results.

You must supply the command line to be used. There is no default.

```
from buildbot.steps.python import PyLint

...
f.addStep(PyLint(command=["pylint", "src"]))
```

6.1.8 Transferring Files

Most of the work involved in a build will take place on the buildslave. But occasionally it is useful to do some work on the buildmaster side. The most basic way to involve the buildmaster is simply to move a file from the slave to the master, or vice versa. There are a pair of BuildSteps named `FileUpload` and `FileDownload` to provide this functionality. `FileUpload` moves a file *up to* the master, while `FileDownload` moves a file *down from* the master.

As an example, let's assume that there is a step which produces an HTML file within the source tree that contains some sort of generated project documentation. We want to move this file to the buildmaster, into a '`~/public_html`' directory, so it can be visible to developers. This file will wind up in the slave-side working directory

under the name 'docs/reference.html'. We want to put it into the master-side '~/public_html/ref.html'.

```
from buildbot.steps.shell import ShellCommand
from buildbot.steps.transfer import FileUpload

f.addStep(ShellCommand(command=["make", "docs"]))
f.addStep(FileUpload(slavesrc="docs/reference.html",
                     masterdest="~/public_html/ref.html"))
```

The masterdest= argument will be passed to os.path.expanduser, so things like "~" will be expanded properly. Non-absolute paths will be interpreted relative to the buildmaster's base directory. Likewise, the slavesrc= argument will be expanded and interpreted relative to the builder's working directory.

To move a file from the master to the slave, use the FileDownload command. For example, let's assume that some step requires a configuration file that, for whatever reason, could not be recorded in the source code repository or generated on the buildslave side:

```
from buildbot.steps.shell import ShellCommand
from buildbot.steps.transfer import FileDownload

f.addStep(FileDownload(mastersrc="~/todays_build_config.txt",
                       slavedest="build_config.txt"))
f.addStep(ShellCommand(command=["make", "config"]))
```

Like FileUpload, the mastersrc= argument is interpreted relative to the buildmaster's base directory, and the slavedest= argument is relative to the builder's working directory. If the buildslave is running in '~buildslave', and the builder's "builddir" is something like 'tests-i386', then the workdir is going to be '~buildslave/tests-i386/build', and a slavedest= of 'foo/bar.html' will get put in '~buildslave/tests-i386/build/foo/bar.html'. Both of these commands will create any missing intervening directories.

Other Parameters

The maxsize= argument lets you set a maximum size for the file to be transferred. This may help to avoid surprises: transferring a 100MB coredump when you were expecting to move a 10kB status file might take an awfully long time. The blocksize= argument controls how the file is sent over the network: larger blocksizes are slightly more efficient but also consume more memory on each end, and there is a hard-coded limit of about 640kB.

The mode= argument allows you to control the access permissions of the target file, traditionally expressed as an octal integer. The most common value is probably 0755, which sets the "x" executable bit on the file (useful for shell scripts and the like). The default value for mode= is None, which means the permission bits will default to whatever the umask of the writing process is. The default umask tends to be fairly restrictive, but at least on the buildslave you can make it less restrictive with a –umask command-line option at creation time (see Section 2.5.1 [Buildslave Options], page 12).

Transfering Directories

To transfer complete directories from the buildslave to the master, there is a BuildStep named DirectoryUpload. It works like FileUpload, just for directories. However it does

not support the `maxsize`, `blocksize` and `mode` arguments. As an example, let's assume an generated project documentation, which consists of many files (like the output of doxygen or epydoc). We want to move the entire documentation to the buildmaster, into a `~/public_html/docs` directory. On the slave-side the directory can be found under `docs`:

```
from buildbot.steps.shell import ShellCommand
from buildbot.steps.transfer import DirectoryUpload

f.addStep(ShellCommand(command=["make", "docs"]))
f.addStep(DirectoryUpload(slavesrc="docs",
masterdest="~/public_html/docs"))
```

The DirectoryUpload step will create all necessary directories and transfers empty directories, too.

6.1.9 Steps That Run on the Master

Occasionally, it is useful to execute some task on the master, for example to create a directory, deploy a build result, or trigger some other centralized processing. This is possible, in a limited fashion, with the `MasterShellCommand` step.

This step operates similarly to a regular `ShellCommand`, but executes on the master, instead of the slave. To be clear, the enclosing `Build` object must still have a slave object, just as for any other step – only, in this step, the slave does not do anything.

In this example, the step renames a tarball based on the day of the week.

```
from buildbot.steps.transfer import FileUpload
from buildbot.steps.master import MasterShellCommand

f.addStep(FileUpload(slavesrc="widgetsoft.tar.gz",
                     masterdest="/var/buildoutputs/widgetsoft-new.tar.gz"))
f.addStep(MasterShellCommand(command="""
    cd /var/buildoutputs;
    mv widgetsoft-new.tar.gz widgetsoft-`date +%a`.tar.gz"""))
```

6.1.10 Triggering Schedulers

The counterpart to the Triggerable described in section see Section 4.5.7 [Triggerable Scheduler], page 37 is the Trigger BuildStep.

```
from buildbot.steps.trigger import Trigger
f.addStep(Trigger(schedulerNames=['build-prep'],
                  waitForFinish=True,
                  updateSourceStamp=True,
                  set_properties={ 'quick' : False },
                  copy_properties=[ 'release_code_name' ]))
```

The `schedulerNames=` argument lists the Triggerables that should be triggered when this step is executed. Note that it is possible, but not advisable, to create a cycle where a build continually triggers itself, because the schedulers are specified by name.

If `waitForFinish` is True, then the step will not finish until all of the builds from the triggered schedulers have finished. If this argument is False (the default) or not given, then the buildstep succeeds immediately after triggering the schedulers.

If `updateSourceStamp` is True (the default), then step updates the SourceStamp given to the Triggerables to include `got_revision` (the revision actually used in this build) as `revision` (the revision to use in the triggered builds). This is useful to ensure that all of the builds use exactly the same SourceStamp, even if other Changes have occurred while the build was running.

Two parameters allow control of the properties that are passed to the triggered scheduler. To simply copy properties verbatim, list them in the `copy_properties` parameter. To set properties explicitly, use the more sophisticated `set_properties`, which takes a dictionary mapping property names to values. You may use `WithProperties` here to dynamically construct new property values.

6.1.11 Writing New BuildSteps

While it is a good idea to keep your build process self-contained in the source code tree, sometimes it is convenient to put more intelligence into your Buildbot configuration. One way to do this is to write a custom BuildStep. Once written, this Step can be used in the '`master.cfg`' file.

The best reason for writing a custom BuildStep is to better parse the results of the command being run. For example, a BuildStep that knows about JUnit could look at the logfiles to determine which tests had been run, how many passed and how many failed, and then report more detailed information than a simple `rc==0` -based "good/bad" decision.

6.1.11.1 Writing BuildStep Constructors

BuildStep classes have some extra equipment, because they are their own factories. Consider the use of a BuildStep in '`master.cfg`':

```
f.addStep(MyStep(someopt="stuff", anotheropt=1))
```

This creates a single instance of class `MyStep`. However, Buildbot needs a new object each time the step is executed. this is accomplished by storing the information required to instantiate a new object in the `factory` attribute. When the time comes to construct a new Build, BuildFactory consults this attribute (via `getStepFactory`) and instantiates a new step object.

When writing a new step class, then, keep in mind are that you cannot do anything "interesting" in the constructor – limit yourself to checking and storing arguments. To ensure that these arguments are provided to any new objects, call `self.addFactoryArguments` with any keyword arguments your constructor needs.

Keep a `**kwargs` argument on the end of your options, and pass that up to the parent class's constructor.

The whole thing looks like this:

```
class Frobinfy(LoggingBuildStep):
    def __init__(self,
            frob_what="frobee",
            frob_how_many=None,
            frob_how=None,
            **kwargs)

        # check
```

```
            if frob_how_many is None:
                raise TypeError("Frobinfy argument how_many is required")

            # call parent
            LoggingBuildStep.__init__(self, **kwargs)

            # and record arguments for later
            self.addFactoryArguments(
                frob_what=frob_what,
                frob_how_many=frob_how_many,
                frob_how=frob_how)

    class FastFrobnify(Frobnify):
        def __init__(self,
                speed=5,
                **kwargs)
            Frobnify.__init__(self, **kwargs)
            self.addFactoryArguments(
                speed=speed)
```

6.1.11.2 BuildStep LogFiles

Each BuildStep has a collection of "logfiles". Each one has a short name, like "stdio" or "warnings". Each LogFile contains an arbitrary amount of text, usually the contents of some output file generated during a build or test step, or a record of everything that was printed to stdout/stderr during the execution of some command.

These LogFiles are stored to disk, so they can be retrieved later.

Each can contain multiple "channels", generally limited to three basic ones: stdout, stderr, and "headers". For example, when a ShellCommand runs, it writes a few lines to the "headers" channel to indicate the exact argv strings being run, which directory the command is being executed in, and the contents of the current environment variables. Then, as the command runs, it adds a lot of "stdout" and "stderr" messages. When the command finishes, a final "header" line is added with the exit code of the process.

Status display plugins can format these different channels in different ways. For example, the web page shows LogFiles as text/html, with header lines in blue text, stdout in black, and stderr in red. A different URL is available which provides a text/plain format, in which stdout and stderr are collapsed together, and header lines are stripped completely. This latter option makes it easy to save the results to a file and run `grep` or whatever against the output.

Each BuildStep contains a mapping (implemented in a python dictionary) from LogFile name to the actual LogFile objects. Status plugins can get a list of LogFiles to display, for example, a list of HREF links that, when clicked, provide the full contents of the LogFile.

Using LogFiles in custom BuildSteps

The most common way for a custom BuildStep to use a LogFile is to summarize the results of a ShellCommand (after the command has finished running). For example, a compile

step with thousands of lines of output might want to create a summary of just the warning messages. If you were doing this from a shell, you would use something like:

```
grep "warning:" output.log >warnings.log
```

In a custom BuildStep, you could instead create a "warnings" LogFile that contained the same text. To do this, you would add code to your `createSummary` method that pulls lines from the main output log and creates a new LogFile with the results:

```
def createSummary(self, log):
    warnings = []
    for line in log.readlines():
        if "warning:" in line:
            warnings.append()
    self.addCompleteLog('warnings', "".join(warnings))
```

This example uses the `addCompleteLog` method, which creates a new LogFile, puts some text in it, and then "closes" it, meaning that no further contents will be added. This LogFile will appear in the HTML display under an HREF with the name "warnings", since that is the name of the LogFile.

You can also use `addHTMLLog` to create a complete (closed) LogFile that contains HTML instead of plain text. The normal LogFile will be HTML-escaped if presented through a web page, but the HTML LogFile will not. At the moment this is only used to present a pretty HTML representation of an otherwise ugly exception traceback when something goes badly wrong during the BuildStep.

In contrast, you might want to create a new LogFile at the beginning of the step, and add text to it as the command runs. You can create the LogFile and attach it to the build by calling `addLog`, which returns the LogFile object. You then add text to this LogFile by calling methods like `addStdout` and `addHeader`. When you are done, you must call the `finish` method so the LogFile can be closed. It may be useful to create and populate a LogFile like this from a LogObserver method See Section 6.1.11.4 [Adding LogObservers], page 91.

The `logfiles=` argument to `ShellCommand` (see see Section 6.1.4 [ShellCommand], page 78) creates new LogFiles and fills them in realtime by asking the buildslave to watch a actual file on disk. The buildslave will look for additions in the target file and report them back to the BuildStep. These additions will be added to the LogFile by calling `addStdout`. These secondary LogFiles can be used as the source of a LogObserver just like the normal "stdio" LogFile.

6.1.11.3 Reading Logfiles

Once a LogFile has been added to a BuildStep with `addLog()`, `addCompleteLog()`, `addHTMLLog()`, or `logfiles=`, your BuildStep can retrieve it by using `getLog()`:

```
class MyBuildStep(ShellCommand):
    logfiles = { "nodelog": "_test/node.log" }

    def evaluateCommand(self, cmd):
        nodelog = self.getLog("nodelog")
        if "STARTED" in nodelog.getText():
            return SUCCESS
```

```
        else:
            return FAILURE
```

For a complete list of the methods you can call on a LogFile, please see the docstrings on the `IStatusLog` class in '`buildbot/interfaces.py`'.

6.1.11.4 Adding LogObservers

Most shell commands emit messages to stdout or stderr as they operate, especially if you ask them nicely with a `--verbose` flag of some sort. They may also write text to a log file while they run. Your BuildStep can watch this output as it arrives, to keep track of how much progress the command has made. You can get a better measure of progress by counting the number of source files compiled or test cases run than by merely tracking the number of bytes that have been written to stdout. This improves the accuracy and the smoothness of the ETA display.

To accomplish this, you will need to attach a `LogObserver` to one of the log channels, most commonly to the "stdio" channel but perhaps to another one which tracks a log file. This observer is given all text as it is emitted from the command, and has the opportunity to parse that output incrementally. Once the observer has decided that some event has occurred (like a source file being compiled), it can use the `setProgress` method to tell the BuildStep about the progress that this event represents.

There are a number of pre-built `LogObserver` classes that you can choose from (defined in `buildbot.process.buildstep`, and of course you can subclass them to add further customization. The `LogLineObserver` class handles the grunt work of buffering and scanning for end-of-line delimiters, allowing your parser to operate on complete stdout/stderr lines. (Lines longer than a set maximum length are dropped; the maximum defaults to 16384 bytes, but you can change it by calling `setMaxLineLength()` on your `LogLineObserver` instance. Use `sys.maxint` for effective infinity.)

For example, let's take a look at the `TrialTestCaseCounter`, which is used by the Trial step to count test cases as they are run. As Trial executes, it emits lines like the following:

```
buildbot.test.test_config.ConfigTest.testDebugPassword ... [OK]
buildbot.test.test_config.ConfigTest.testEmpty ... [OK]
buildbot.test.test_config.ConfigTest.testIRC ... [FAIL]
buildbot.test.test_config.ConfigTest.testLocks ... [OK]
```

When the tests are finished, trial emits a long line of "======" and then some lines which summarize the tests that failed. We want to avoid parsing these trailing lines, because their format is less well-defined than the "[OK]" lines.

The parser class looks like this:

```
from buildbot.process.buildstep import LogLineObserver

class TrialTestCaseCounter(LogLineObserver):
    _line_re = re.compile(r'^([\w\.]+) \.\.\. \[([^\]]+)\]$')
    numTests = 0
    finished = False

    def outLineReceived(self, line):
        if self.finished:
```

```
            return
        if line.startswith("=" * 40):
            self.finished = True
            return

        m = self._line_re.search(line.strip())
        if m:
            testname, result = m.groups()
            self.numTests += 1
            self.step.setProgress('tests', self.numTests)
```

This parser only pays attention to stdout, since that's where trial writes the progress lines. It has a mode flag named `finished` to ignore everything after the "====" marker, and a scary-looking regular expression to match each line while hopefully ignoring other messages that might get displayed as the test runs.

Each time it identifies a test has been completed, it increments its counter and delivers the new progress value to the step with `self.step.setProgress`. This class is specifically measuring progress along the "tests" metric, in units of test cases (as opposed to other kinds of progress like the "output" metric, which measures in units of bytes). The Progress-tracking code uses each progress metric separately to come up with an overall completion percentage and an ETA value.

To connect this parser into the `Trial` BuildStep, `Trial.__init__` ends with the following clause:

```
# this counter will feed Progress along the 'test cases' metric
counter = TrialTestCaseCounter()
self.addLogObserver('stdio', counter)
self.progressMetrics += ('tests',)
```

This creates a TrialTestCaseCounter and tells the step that the counter wants to watch the "stdio" log. The observer is automatically given a reference to the step in its `.step` attribute.

A Somewhat Whimsical Example

Let's say that we've got some snazzy new unit-test framework called Framboozle. It's the hottest thing since sliced bread. It slices, it dices, it runs unit tests like there's no tomorrow. Plus if your unit tests fail, you can use its name for a Web 2.1 startup company, make millions of dollars, and hire engineers to fix the bugs for you, while you spend your afternoons lazily hang-gliding along a scenic pacific beach, blissfully unconcerned about the state of your tests.[2]

To run a Framboozle-enabled test suite, you just run the 'framboozler' command from the top of your source code tree. The 'framboozler' command emits a bunch of stuff to stdout, but the most interesting bit is that it emits the line "FNURRRGH!" every time it finishes running a test case[3]. You'd like to have a test-case counting LogObserver that watches for these lines and counts them, because counting them will help the buildbot more accurately calculate how long the build will take, and this will let you know exactly how

[2] framboozle.com is still available. Remember, I get 10% :).

[3] Framboozle gets very excited about running unit tests.

long you can sneak out of the office for your hang-gliding lessons without anyone noticing that you're gone.

This will involve writing a new BuildStep (probably named "Framboozle") which inherits from ShellCommand. The BuildStep class definition itself will look something like this:

```
# START
from buildbot.steps.shell import ShellCommand
from buildbot.process.buildstep import LogLineObserver

class FNURRRGHCounter(LogLineObserver):
    numTests = 0
    def outLineReceived(self, line):
        if "FNURRRGH!" in line:
            self.numTests += 1
            self.step.setProgress('tests', self.numTests)

class Framboozle(ShellCommand):
    command = ["framboozler"]

    def __init__(self, **kwargs):
        ShellCommand.__init__(self, **kwargs)   # always upcall!
        counter = FNURRRGHCounter())
        self.addLogObserver('stdio', counter)
        self.progressMetrics += ('tests',)
# FINISH
```

So that's the code that we want to wind up using. How do we actually deploy it?

You have a couple of different options.

Option 1: The simplest technique is to simply put this text (everything from START to FINISH) in your master.cfg file, somewhere before the BuildFactory definition where you actually use it in a clause like:

```
f = BuildFactory()
f.addStep(SVN(svnurl="stuff"))
f.addStep(Framboozle())
```

Remember that master.cfg is secretly just a python program with one job: populating the BuildmasterConfig dictionary. And python programs are allowed to define as many classes as they like. So you can define classes and use them in the same file, just as long as the class is defined before some other code tries to use it.

This is easy, and it keeps the point of definition very close to the point of use, and whoever replaces you after that unfortunate hang-gliding accident will appreciate being able to easily figure out what the heck this stupid "Framboozle" step is doing anyways. The downside is that every time you reload the config file, the Framboozle class will get redefined, which means that the buildmaster will think that you've reconfigured all the Builders that use it, even though nothing changed. Bleh.

Option 2: Instead, we can put this code in a separate file, and import it into the master.cfg file just like we would the normal buildsteps like ShellCommand and SVN.

Create a directory named ~/lib/python, put everything from START to FINISH in ~/lib/python/framboozle.py, and run your buildmaster using:

```
PYTHONPATH=~/lib/python buildbot start MASTERDIR
```

or use the 'Makefile.buildbot' to control the way `buildbot start` works. Or add something like this to something like your ~/.bashrc or ~/.bash_profile or ~/.cshrc:

```
export PYTHONPATH=~/lib/python
```

Once we've done this, our master.cfg can look like:

```
from framboozle import Framboozle
f = BuildFactory()
f.addStep(SVN(svnurl="stuff"))
f.addStep(Framboozle())
```

or:

```
import framboozle
f = BuildFactory()
f.addStep(SVN(svnurl="stuff"))
f.addStep(framboozle.Framboozle())
```

(check out the python docs for details about how "import" and "from A import B" work).

What we've done here is to tell python that every time it handles an "import" statement for some named module, it should look in our ~/lib/python/ for that module before it looks anywhere else. After our directories, it will try in a bunch of standard directories too (including the one where buildbot is installed). By setting the PYTHONPATH environment variable, you can add directories to the front of this search list.

Python knows that once it "import"s a file, it doesn't need to re-import it again. This means that reconfiguring the buildmaster (with "buildbot reconfig", for example) won't make it think the Framboozle class has changed every time, so the Builders that use it will not be spuriously restarted. On the other hand, you either have to start your buildmaster in a slightly weird way, or you have to modify your environment to set the PYTHONPATH variable.

Option 3: Install this code into a standard python library directory

Find out what your python's standard include path is by asking it:

```
80:warner@luther% python
Python 2.4.4c0 (#2, Oct  2 2006, 00:57:46)
[GCC 4.1.2 20060928 (prerelease) (Debian 4.1.1-15)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> import pprint
>>> pprint.pprint(sys.path)
['',
 '/usr/lib/python24.zip',
 '/usr/lib/python2.4',
 '/usr/lib/python2.4/plat-linux2',
 '/usr/lib/python2.4/lib-tk',
 '/usr/lib/python2.4/lib-dynload',
```

```
'/usr/local/lib/python2.4/site-packages',
'/usr/lib/python2.4/site-packages',
'/usr/lib/python2.4/site-packages/Numeric',
'/var/lib/python-support/python2.4',
'/usr/lib/site-python']
```

In this case, putting the code into /usr/local/lib/python2.4/site-packages/framboozle.py would work just fine. We can use the same master.cfg "import framboozle" statement as in Option 2. By putting it in a standard include directory (instead of the decidedly non-standard ~/lib/python), we don't even have to set PYTHONPATH to anything special. The downside is that you probably have to be root to write to one of those standard include directories.

Option 4: Submit the code for inclusion in the Buildbot distribution

Make a fork of buildbot on http://github.com/djmitche/buildbot or post a patch in a bug at http://buildbot.net. In either case, post a note about your patch to the mailing list, so others can provide feedback and, eventually, commit it.

```
from buildbot.steps import framboozle
f = BuildFactory()
f.addStep(SVN(svnurl="stuff"))
f.addStep(framboozle.Framboozle())
```

And then you don't even have to install framboozle.py anywhere on your system, since it will ship with Buildbot. You don't have to be root, you don't have to set PYTHONPATH. But you do have to make a good case for Framboozle being worth going into the main distribution, you'll probably have to provide docs and some unit test cases, you'll need to figure out what kind of beer the author likes, and then you'll have to wait until the next release. But in some environments, all this is easier than getting root on your buildmaster box, so the tradeoffs may actually be worth it.

Putting the code in master.cfg (1) makes it available to that buildmaster instance. Putting it in a file in a personal library directory (2) makes it available for any buildmasters you might be running. Putting it in a file in a system-wide shared library directory (3) makes it available for any buildmasters that anyone on that system might be running. Getting it into the buildbot's upstream repository (4) makes it available for any buildmasters that anyone in the world might be running. It's all a matter of how widely you want to deploy that new class.

6.1.11.5 BuildStep URLs

Each BuildStep has a collection of "links". Like its collection of LogFiles, each link has a name and a target URL. The web status page creates HREFs for each link in the same box as it does for LogFiles, except that the target of the link is the external URL instead of an internal link to a page that shows the contents of the LogFile.

These external links can be used to point at build information hosted on other servers. For example, the test process might produce an intricate description of which tests passed and failed, or some sort of code coverage data in HTML form, or a PNG or GIF image with a graph of memory usage over time. The external link can provide an easy way for users to navigate from the buildbot's status page to these external web sites or file servers. Note that the step itself is responsible for insuring that there will be a document available at the given

URL (perhaps by using `scp` to copy the HTML output to a '`~/public_html/`' directory on a remote web server). Calling `addURL` does not magically populate a web server.

To set one of these links, the BuildStep should call the `addURL` method with the name of the link and the target URL. Multiple URLs can be set.

In this example, we assume that the `make test` command causes a collection of HTML files to be created and put somewhere on the coverage.example.org web server, in a filename that incorporates the build number.

```
class TestWithCodeCoverage(BuildStep):
    command = ["make", "test",
               WithProperties("buildnum=%s", "buildnumber")]

    def createSummary(self, log):
        buildnumber = self.getProperty("buildnumber")
        url = "http://coverage.example.org/builds/%s.html" % buildnumber
        self.addURL("coverage", url)
```

You might also want to extract the URL from some special message output by the build process itself:

```
class TestWithCodeCoverage(BuildStep):
    command = ["make", "test",
               WithProperties("buildnum=%s", "buildnumber")]

    def createSummary(self, log):
        output = StringIO(log.getText())
        for line in output.readlines():
            if line.startswith("coverage-url:"):
                url = line[len("coverage-url:"):].strip()
                self.addURL("coverage", url)
                return
```

Note that a build process which emits both stdout and stderr might cause this line to be split or interleaved between other lines. It might be necessary to restrict the getText() call to only stdout with something like this:

```
        output = StringIO("".join([c[1]
                                   for c in log.getChunks()
                                   if c[0] == LOG_CHANNEL_STDOUT]))
```

Of course if the build is run under a PTY, then stdout and stderr will be merged before the buildbot ever sees them, so such interleaving will be unavoidable.

6.2 Interlocks

Until now, we assumed that a master can run builds at any slave whenever needed or desired. Some times, you want to enforce additional constraints on builds. For reasons like limited network bandwidth, old slave machines, or a self-willed data base server, you may want to limit the number of builds (or build steps) that can access a resource.

The mechanism used by Buildbot is known as the read/write lock.[4] It allows either many readers or a single writer but not a combination of readers and writers. The general lock has been modified and extended for use in Buildbot. Firstly, the general lock allows an infinite number of readers. In Buildbot, we often want to put an upper limit on the number of readers, for example allowing two out of five possible builds at the same time. To do this, the lock counts the number of active readers. Secondly, the terms *read mode* and *write mode* are confusing in Buildbot context. They have been replaced by *counting mode* (since the lock counts them) and *exclusive mode*. As a result of these changes, locks in Buildbot allow a number of builds (upto some fixed number) in counting mode, or they allow one build in exclusive mode.

Often, not all slaves are equal. To allow for this situation, Buildbot allows to have a separate upper limit on the count for each slave. In this way, you can have at most 3 concurrent builds at a fast slave, 2 at a slightly older slave, and 1 at all other slaves.

The final thing you can specify when you introduce a new lock is its scope. Some constraints are global – they must be enforced over all slaves. Other constraints are local to each slave. A *master lock* is used for the global constraints. You can ensure for example that at most one build (of all builds running at all slaves) accesses the data base server. With a *slave lock* you can add a limit local to each slave. With such a lock, you can for example enforce an upper limit to the number of active builds at a slave, like above.

Time for a few examples. Below a master lock is defined to protect a data base, and a slave lock is created to limit the number of builds at each slave.

```
from buildbot import locks

db_lock = locks.MasterLock("database")
build_lock = locks.SlaveLock("slave_builds",
                             maxCount = 1,
                             maxCountForSlave = { 'fast': 3, 'new': 2 })
```

After importing locks from buildbot, `db_lock` is defined to be a master lock. The `"database"` string is used for uniquely identifying the lock. At the next line, a slave lock called `build_lock` is created. It is identified by the `"slave_builds"` string. Since the requirements of the lock are a bit more complicated, two optional arguments are also specified. The `maxCount` parameter sets the default limit for builds in counting mode to `1`. For the slave called `'fast'` however, we want to have at most three builds, and for the slave called `'new'` the upper limit is two builds running at the same time.

The next step is using the locks in builds. Buildbot allows a lock to be used during an entire build (from beginning to end), or only during a single build step. In the latter case, the lock is claimed for use just before the step starts, and released again when the step ends. To prevent deadlocks,[5] it is not possible to claim or release locks at other times.

To use locks, you should add them with a `locks` argument. Each use of a lock is either in counting mode (that is, possibly shared with other builds) or in exclusive mode. A build

[4] See http://en.wikipedia.org/wiki/Read/write_lock_pattern for more information.

[5] Deadlock is the situation where two or more slaves each hold a lock in exclusive mode, and in addition want to claim the lock held by the other slave exclusively as well. Since locks allow at most one exclusive user, both slaves will wait forever.

or build step proceeds only when it has acquired all locks. If a build or step needs a lot of locks, it may be starved[6] by other builds that need fewer locks.

To illustrate use of locks, a few examples.

```
from buildbot import locks
from buildbot.steps import source, shell
from buildbot.process import factory

db_lock = locks.MasterLock("database")
build_lock = locks.SlaveLock("slave_builds",
                             maxCount = 1,
                             maxCountForSlave = { 'fast': 3, 'new': 2 })

f = factory.BuildFactory()
f.addStep(source.SVN(svnurl="http://example.org/svn/Trunk"))
f.addStep(shell.ShellCommand(command="make all"))
f.addStep(shell.ShellCommand(command="make test",
                             locks=[db_lock.access('exclusive')]))

b1 = {'name': 'full1', 'slavename': 'fast',  'builddir': 'f1', 'factory': f,
       'locks': [build_lock.access('counting')] }

b2 = {'name': 'full2', 'slavename': 'new',   'builddir': 'f2', 'factory': f.
       'locks': [build_lock.access('counting')] }

b3 = {'name': 'full3', 'slavename': 'old',   'builddir': 'f3', 'factory': f.
       'locks': [build_lock.access('counting')] }

b4 = {'name': 'full4', 'slavename': 'other', 'builddir': 'f4', 'factory': f.
       'locks': [build_lock.access('counting')] }

c['builders'] = [b1, b2, b3, b4]
```

Here we have four slaves `b1`, `b2`, `b3`, and `b4`. Each slave performs the same checkout, make, and test build step sequence. We want to enforce that at most one test step is executed between all slaves due to restrictions with the data base server. This is done by adding the `locks=` parameter with the third step. It takes a list of locks with their access mode. In this case only the `db_lock` is needed. The exclusive access mode is used to ensure there is at most one slave that executes the test step.

In addition to exclusive accessing the data base, we also want slaves to stay responsive even under the load of a large number of builds being triggered. For this purpose, the slave lock called `build_lock` is defined. Since the restraint holds for entire builds, the lock is specified in the builder with `'locks': [build_lock.access('counting')]`.

[6] Starving is the situation that only a few locks are available, and they are immediately grabbed by another build. As a result, it may take a long time before all locks needed by the starved build are free at the same time.

6.3 Build Factories

Each Builder is equipped with a "build factory", which is responsible for producing the actual `Build` objects that perform each build. This factory is created in the configuration file, and attached to a Builder through the `factory` element of its dictionary.

The standard `BuildFactory` object creates `Build` objects by default. These Builds will each execute a collection of BuildSteps in a fixed sequence. Each step can affect the results of the build, but in general there is little intelligence to tie the different steps together. You can create subclasses of `Build` to implement more sophisticated build processes, and then use a subclass of `BuildFactory` (or simply set the `buildClass` attribute) to create instances of your new Build subclass.

6.3.1 BuildStep Objects

The steps used by these builds are all subclasses of `BuildStep`. The standard ones provided with Buildbot are documented later, See Section 6.1 [Build Steps], page 66. You can also write your own subclasses to use in builds.

The basic behavior for a `BuildStep` is to:

- run for a while, then stop
- possibly invoke some RemoteCommands on the attached build slave
- possibly produce a set of log files
- finish with a status described by one of four values defined in buildbot.status.builder: SUCCESS, WARNINGS, FAILURE, SKIPPED
- provide a list of short strings to describe the step
- define a color (generally green, orange, or red) with which the step should be displayed

More sophisticated steps may produce additional information and provide it to later build steps, or store it in the factory to provide to later builds.

6.3.2 BuildFactory

The default `BuildFactory`, provided in the `buildbot.process.factory` module, contains an internal list of "BuildStep specifications": a list of `(step_class, kwargs)` tuples for each. These specification tuples are constructed when the config file is read, by asking the instances passed to `addStep` for their subclass and arguments.

When asked to create a Build, the `BuildFactory` puts a copy of the list of step specifications into the new Build object. When the Build is actually started, these step specifications are used to create the actual set of BuildSteps, which are then executed one at a time. This serves to give each Build an independent copy of each step. For example, a build which consists of a CVS checkout followed by a `make build` would be constructed as follows:

```
from buildbot.steps import source, shell
from buildbot.process import factory

f = factory.BuildFactory()
f.addStep(source.CVS(cvsroot=CVSROOT, cvsmodule="project", mode="update"))
f.addStep(shell.Compile(command=["make", "build"]))
```

(To support config files from buildbot-0.7.5 and earlier, `addStep` also accepts the `f.addStep(shell.Compile, command=["make","build"])` form, although its use is

discouraged because then the `Compile` step doesn't get to validate or complain about its arguments until build time. The modern pass-by-instance approach allows this validation to occur while the config file is being loaded, where the admin has a better chance of noticing problems).

It is also possible to pass a list of steps into the `BuildFactory` when it is created. Using `addStep` is usually simpler, but there are cases where is is more convenient to create the list of steps ahead of time.:

```
from buildbot.steps import source, shell
from buildbot.process import factory

all_steps = [source.CVS(cvsroot=CVSROOT, cvsmodule="project", mode="update"),
             shell.Compile(command=["make", "build"]),
            ]
f = factory.BuildFactory(all_steps)
```

Each step can affect the build process in the following ways:

- If the step's `haltOnFailure` attribute is True, then a failure in the step (i.e. if it completes with a result of FAILURE) will cause the whole build to be terminated immediately: no further steps will be executed, with the exception of steps with `alwaysRun` set to True. `haltOnFailure` is useful for setup steps upon which the rest of the build depends: if the CVS checkout or `./configure` process fails, there is no point in trying to compile or test the resulting tree.
- If the step's `alwaysRun` attribute is True, then it will always be run, regardless of if previous steps have failed. This is useful for cleanup steps that should always be run to return the build directory or build slave into a good state.
- If the `flunkOnFailure` or `flunkOnWarnings` flag is set, then a result of FAILURE or WARNINGS will mark the build as a whole as FAILED. However, the remaining steps will still be executed. This is appropriate for things like multiple testing steps: a failure in any one of them will indicate that the build has failed, however it is still useful to run them all to completion.
- Similarly, if the `warnOnFailure` or `warnOnWarnings` flag is set, then a result of FAILURE or WARNINGS will mark the build as having WARNINGS, and the remaining steps will still be executed. This may be appropriate for certain kinds of optional build or test steps. For example, a failure experienced while building documentation files should be made visible with a WARNINGS result but not be serious enough to warrant marking the whole build with a FAILURE.

In addition, each Step produces its own results, may create logfiles, etc. However only the flags described above have any effect on the build as a whole.

The pre-defined BuildSteps like `CVS` and `Compile` have reasonably appropriate flags set on them already. For example, without a source tree there is no point in continuing the build, so the `CVS` class has the `haltOnFailure` flag set to True. Look in '`buildbot/steps/*.py`' to see how the other Steps are marked.

Each Step is created with an additional `workdir` argument that indicates where its actions should take place. This is specified as a subdirectory of the slave builder's base directory, with a default value of `build`. This is only implemented as a step argument (as

opposed to simply being a part of the base directory) because the CVS/SVN steps need to perform their checkouts from the parent directory.

6.3.2.1 BuildFactory Attributes

Some attributes from the BuildFactory are copied into each Build.

`useProgress`
: (defaults to True): if True, the buildmaster keeps track of how long each step takes, so it can provide estimates of how long future builds will take. If builds are not expected to take a consistent amount of time (such as incremental builds in which a random set of files are recompiled or tested each time), this should be set to False to inhibit progress-tracking.

6.3.2.2 Quick builds

The difference between a "full build" and a "quick build" is that quick builds are generally done incrementally, starting with the tree where the previous build was performed. That simply means that the source-checkout step should be given a `mode='update'` flag, to do the source update in-place.

In addition to that, the `useProgress` flag should be set to False. Incremental builds will (or at least the ought to) compile as few files as necessary, so they will take an unpredictable amount of time to run. Therefore it would be misleading to claim to predict how long the build will take.

6.3.3 Process-Specific build factories

Many projects use one of a few popular build frameworks to simplify the creation and maintenance of Makefiles or other compilation structures. Buildbot provides several pre-configured BuildFactory subclasses which let you build these projects with a minimum of fuss.

6.3.3.1 GNUAutoconf

GNU Autoconf (`http://www.gnu.org/software/autoconf/`) is a software portability tool, intended to make it possible to write programs in C (and other languages) which will run on a variety of UNIX-like systems. Most GNU software is built using autoconf. It is frequently used in combination with GNU automake. These tools both encourage a build process which usually looks like this:

```
% CONFIG_ENV=foo ./configure --with-flags
% make all
% make check
# make install
```

(except of course the Buildbot always skips the `make install` part).

The Buildbot's `buildbot.process.factory.GNUAutoconf` factory is designed to build projects which use GNU autoconf and/or automake. The configuration environment variables, the configure flags, and command lines used for the compile and test are all configurable, in general the default values will be suitable.

Example:

```
# use the s() convenience function defined earlier
f = factory.GNUAutoconf(source=s(source.SVN, svnurl=URL, mode="copy"),
                        flags=["--disable-nls"])
```

Required Arguments:

`source` This argument must be a step specification tuple that provides a BuildStep to generate the source tree.

Optional Arguments:

`configure`
The command used to configure the tree. Defaults to `./configure`. Accepts either a string or a list of shell argv elements.

`configureEnv`
The environment used for the initial configuration step. This accepts a dictionary which will be merged into the buildslave's normal environment. This is commonly used to provide things like `CFLAGS="-O2 -g"` (to turn off debug symbols during the compile). Defaults to an empty dictionary.

`configureFlags`
A list of flags to be appended to the argument list of the configure command. This is commonly used to enable or disable specific features of the autoconf-controlled package, like `["--without-x"]` to disable windowing support. Defaults to an empty list.

`compile` this is a shell command or list of argv values which is used to actually compile the tree. It defaults to `make all`. If set to None, the compile step is skipped.

`test` this is a shell command or list of argv values which is used to run the tree's self-tests. It defaults to `make check`. If set to None, the test step is skipped.

6.3.3.2 CPAN

Most Perl modules available from the CPAN (`http://www.cpan.org/`) archive use the `MakeMaker` module to provide configuration, build, and test services. The standard build routine for these modules looks like:

```
% perl Makefile.PL
% make
% make test
# make install
```

(except again Buildbot skips the install step)

Buildbot provides a `CPAN` factory to compile and test these projects.

Arguments:

`source` (required): A step specification tuple, like that used by GNUAutoconf.

`perl` A string which specifies the `perl` executable to use. Defaults to just `perl`.

6.3.3.3 Python distutils

Most Python modules use the `distutils` package to provide configuration and build services. The standard build process looks like:

```
% python ./setup.py build
% python ./setup.py install
```

Unfortunately, although Python provides a standard unit-test framework named `unittest`, to the best of my knowledge `distutils` does not provide a standardized target to run such unit tests. (Please let me know if I'm wrong, and I will update this factory.)

The `Distutils` factory provides support for running the build part of this process. It accepts the same `source=` parameter as the other build factories.

Arguments:

`source` (required): A step specification tuple, like that used by GNUAutoconf.

`python` A string which specifies the `python` executable to use. Defaults to just `python`.

`test` Provides a shell command which runs unit tests. This accepts either a string or a list. The default value is None, which disables the test step (since there is no common default command to run unit tests in distutils modules).

6.3.3.4 Python/Twisted/trial projects

Twisted provides a unit test tool named `trial` which provides a few improvements over Python's built-in `unittest` module. Many python projects which use Twisted for their networking or application services also use trial for their unit tests. These modules are usually built and tested with something like the following:

```
% python ./setup.py build
% PYTHONPATH=build/lib.linux-i686-2.3 trial -v PROJECTNAME.test
% python ./setup.py install
```

Unfortunately, the '`build/lib`' directory into which the built/copied .py files are placed is actually architecture-dependent, and I do not yet know of a simple way to calculate its value. For many projects it is sufficient to import their libraries "in place" from the tree's base directory (`PYTHONPATH=.`).

In addition, the *PROJECTNAME* value where the test files are located is project-dependent: it is usually just the project's top-level library directory, as common practice suggests the unit test files are put in the `test` sub-module. This value cannot be guessed, the `Trial` class must be told where to find the test files.

The `Trial` class provides support for building and testing projects which use distutils and trial. If the test module name is specified, trial will be invoked. The library path used for testing can also be set.

One advantage of trial is that the Buildbot happens to know how to parse trial output, letting it identify which tests passed and which ones failed. The Buildbot can then provide fine-grained reports about how many tests have failed, when individual tests fail when they had been passing previously, etc.

Another feature of trial is that you can give it a series of source .py files, and it will search them for special `test-case-name` tags that indicate which test cases provide coverage for

that file. Trial can then run just the appropriate tests. This is useful for quick builds, where you want to only run the test cases that cover the changed functionality.

Arguments:

`testpath` Provides a directory to add to `PYTHONPATH` when running the unit tests, if tests are being run. Defaults to `.` to include the project files in-place. The generated build library is frequently architecture-dependent, but may simply be '`build/lib`' for pure-python modules.

`python` which python executable to use. This list will form the start of the argv array that will launch trial. If you use this, you should set `trial` to an explicit path (like `/usr/bin/trial` or `./bin/trial`). The parameter defaults to `None`, which leaves it out entirely (running `trial args` instead of `python ./bin/trial args`). Likely values are `['python']`, `['python2.2']`, or `['python', '-Wall']`.

`trial` provides the name of the `trial` command. It is occasionally useful to use an alternate executable, such as `trial2.2` which might run the tests under an older version of Python. Defaults to `trial`.

`trialMode`
a list of arguments to pass to trial, specifically to set the reporting mode. This defaults to `['--reporter=bwverbose']`, which only works for Twisted-2.1.0 and later.

`trialArgs`
a list of arguments to pass to trial, available to turn on any extra flags you like. Defaults to `[]`.

`tests` Provides a module name or names which contain the unit tests for this project. Accepts a string, typically `PROJECTNAME.test`, or a list of strings. Defaults to None, indicating that no tests should be run. You must either set this or `testChanges`.

`testChanges`
if True, ignore the `tests` parameter and instead ask the Build for all the files that make up the Changes going into this build. Pass these filenames to trial and ask it to look for test-case-name tags, running just the tests necessary to cover the changes.

`recurse` If `True`, tells Trial (with the `--recurse` argument) to look in all subdirectories for additional test cases.

`reactor` which reactor to use, like 'gtk' or 'java'. If not provided, the Twisted's usual platform-dependent default is used.

`randomly` If `True`, tells Trial (with the `--random=0` argument) to run the test cases in random order, which sometimes catches subtle inter-test dependency bugs. Defaults to `False`.

The step can also take any of the `ShellCommand` arguments, e.g., `haltOnFailure`.

Unless one of `tests` or `testChanges` are set, the step will generate an exception.

7 Status Delivery

More details are available in the docstrings for each class, use a command like `pydoc buildbot.status.html.WebStatus` to see them. Most status delivery objects take a `categories=` argument, which can contain a list of "category" names: in this case, it will only show status for Builders that are in one of the named categories.

(implementor's note: each of these objects should be a service.MultiService which will be attached to the BuildMaster object when the configuration is processed. They should use `self.parent.getStatus()` to get access to the top-level IStatus object, either inside `startService` or later. They may call `status.subscribe()` in `startService` to receive notifications of builder events, in which case they must define `builderAdded` and related methods. See the docstrings in '`buildbot/interfaces.py`' for full details.)

7.1 WebStatus

The `buildbot.status.html.WebStatus` status target runs a small web server inside the buildmaster. You can point a browser at this web server and retrieve information about every build the buildbot knows about, as well as find out what the buildbot is currently working on.

The first page you will see is the "Welcome Page", which contains links to all the other useful pages. This page is simply served from the '`public_html/index.html`' file in the buildmaster's base directory, where it is created by the `buildbot create-master` command along with the rest of the buildmaster.

The most complex resource provided by `WebStatus` is the "Waterfall Display", which shows a time-based chart of events. This somewhat-busy display provides detailed information about all steps of all recent builds, and provides hyperlinks to look at individual build logs and source changes. By simply reloading this page on a regular basis, you will see a complete description of everything the buildbot is currently working on.

A similar, but more developer-oriented display is the `"Grid"` display. This arranges builds by SourceStamp (horizontal axis) and builder (vertical axis), and can provide quick information as to which revisions are passing or failing on which builders.

There are also pages with more specialized information. For example, there is a page which shows the last 20 builds performed by the buildbot, one line each. Each line is a link to detailed information about that build. By adding query arguments to the URL used to reach this page, you can narrow the display to builds that involved certain branches, or which ran on certain Builders. These pages are described in great detail below.

When the buildmaster is created, a subdirectory named '`public_html/`' is created in its base directory. By default, `WebStatus` will serve files from this directory: for example, when a user points their browser at the buildbot's `WebStatus` URL, they will see the contents of the '`public_html/index.html`' file. Likewise, '`public_html/robots.txt`', '`public_html/buildbot.css`', and '`public_html/favicon.ico`' are all useful things to have in there. The first time a buildmaster is created, the '`public_html`' directory is populated with some sample files, which you will probably want to customize for your own project. These files are all static: the buildbot does not modify them in any way as it serves them to HTTP clients.

```
from buildbot.status.html import WebStatus
c['status'].append(WebStatus(8080))
```

Note that the initial robots.txt file has Disallow lines for all of the dynamically-generated buildbot pages, to discourage web spiders and search engines from consuming a lot of CPU time as they crawl through the entire history of your buildbot. If you are running the buildbot behind a reverse proxy, you'll probably need to put the robots.txt file somewhere else (at the top level of the parent web server), and replace the URL prefixes in it with more suitable values.

If you would like to use an alternative root directory, add the `public_html=..` option to the `WebStatus` creation:

```
c['status'].append(WebStatus(8080, public_html="/var/www/buildbot"))
```

In addition, if you are familiar with twisted.web *Resource Trees*, you can write code to add additional pages at places inside this web space. Just use `webstatus.putChild` to place these resources.

The following section describes the special URLs and the status views they provide.

7.1.1 WebStatus Configuration Parameters

The most common way to run a `WebStatus` is on a regular TCP port. To do this, just pass in the TCP port number when you create the `WebStatus` instance; this is called the `http_port` argument:

```
from buildbot.status.html import WebStatus
c['status'].append(WebStatus(8080))
```

The `http_port` argument is actually a "strports specification" for the port that the web server should listen on. This can be a simple port number, or a string like `tcp:8080:interface=127.0.0.1` (to limit connections to the loopback interface, and therefore to clients running on the same host)[1].

If instead (or in addition) you provide the `distrib_port` argument, a twisted.web distributed server will be started either on a TCP port (if `distrib_port` is like `"tcp:12345"`) or more likely on a UNIX socket (if `distrib_port` is like `"unix:/path/to/socket"`).

The `distrib_port` option means that, on a host with a suitably-configured twisted-web server, you do not need to consume a separate TCP port for the buildmaster's status web page. When the web server is constructed with `mktap web --user`, URLs that point to `http://host/~username/` are dispatched to a sub-server that is listening on a UNIX socket at `~username/.twisted-web-pb`. On such a system, it is convenient to create a dedicated `buildbot` user, then set `distrib_port` to `"unix:"+os.path.expanduser("~/.twistd-web-pb")`. This configuration will make the HTML status page available at `http://host/~buildbot/` . Suitable URL remapping can make it appear at `http://host/buildbot/`, and the right virtual host setup can even place it at `http://buildbot.host/` .

[1] It may even be possible to provide SSL access by using a specification like `"ssl:12345:privateKey=mykey.pen:certKey=cer` but this is completely untested

7.1.2 Enabling the "Force Build" Button

An important `WebStatus` argument is `allowForce`. If set to True, then the web page will provide a "Force Build" button that allows visitors to manually trigger builds. This is useful for developers to re-run builds that have failed because of intermittent problems in the test suite, or because of libraries that were not installed at the time of the previous build. The default value is False, since it also allows anyone with a web browser to waste computing resources forcing unnecessary builds.

If you do not wish to allow strangers to cause a build to run or to stop current builds, pass an instance of `status.web.auth.IAuth` as a `auth` keyword argument to `WebStatus`. The class `BasicAuth` implements a basic authentication mechanism using a list of user/password tuples provided from the configuration file. The class `HTPasswdAuth` implements an authentication against an '`.htpasswd`' file.

```
from buildbot.status.html import WebStatus
from buildbot.status.web.auth import BasicAuth
users = [('bob', 'secret-pass'), ('jill', 'super-pass')]
c['status'].append(WebStatus(http_port=8080, auth=BasicAuth(users)))

from buildbot.status.web.auth import HTPasswdAuth
file = '/path/to/file'
c['status'].append(WebStatus(http_port=8080, auth=(HTPasswdAuth(file))))
```

7.1.3 Buildbot Web Resources

Certain URLs are "magic", and the pages they serve are created by code in various classes in the '`buildbot.status.web`' package instead of being read from disk. The most common way to access these pages is for the buildmaster admin to write or modify the '`index.html`' page to contain links to them. Of course other project web pages can contain links to these buildbot pages as well.

Many pages can be modified by adding query arguments to the URL. For example, a page which shows the results of the most recent build normally does this for all builders at once. But by appending "?builder=i386" to the end of the URL, the page will show only the results for the "i386" builder. When used in this way, you can add multiple "builder=" arguments to see multiple builders. Remembering that URL query arguments are separated *from each other* with ampersands, a URL that ends in "?builder=i386&builder=ppc" would show builds for just those two Builders.

The `branch=` query argument can be used on some pages. This filters the information displayed by that page down to only the builds or changes which involved the given branch. Use `branch=trunk` to reference the trunk: if you aren't intentionally using branches, you're probably using trunk. Multiple `branch=` arguments can be used to examine multiple branches at once (so appending `?branch=foo&branch=bar` to the URL will show builds involving either branch). No `branch=` arguments means to show builds and changes for all branches.

Some pages may include the Builder name or the build number in the main part of the URL itself. For example, a page that describes Build #7 of the "i386" builder would live at '`/builders/i386/builds/7`'.

The table below lists all of the internal pages and the URLs that can be used to access them.

NOTE: of the pages described here, `/slave_status_timeline` and `/last_build` have not yet been implemented, and `/xmlrpc` has only a few methods so far. Future releases will improve this.

`/waterfall`

This provides a chronologically-oriented display of the activity of all builders. It is the same display used by the Waterfall display.

By adding one or more "builder=" query arguments, the Waterfall is restricted to only showing information about the given Builders. By adding one or more "branch=" query arguments, the display is restricted to showing information about the given branches. In addition, adding one or more "category=" query arguments to the URL will limit the display to Builders that were defined with one of the given categories.

A 'show_events=true' query argument causes the display to include non-Build events, like slaves attaching and detaching, as well as reconfiguration events. 'show_events=false' hides these events. The default is to show them.

By adding the 'failures_only=true' query argument, the Waterfall is restricted to only showing information about the builders that are currently failing. A builder is considered failing if the last finished build was not successful, a step in the current build(s) is failing, or if the builder is offline.

The `last_time=`, `first_time=`, and `show_time=` arguments will control what interval of time is displayed. The default is to show the latest events, but these can be used to look at earlier periods in history. The `num_events=` argument also provides a limit on the size of the displayed page.

The Waterfall has references to resources many of the other portions of the URL space: '`/builders`' for access to individual builds, '`/changes`' for access to information about source code changes, etc.

`/grid`

This provides a chronologically oriented display of builders, by revision. The builders are listed down the left side of the page, and the revisions are listed across the top.

By adding one ore more "category=" arguments the grid will be restricted to revisions in those categories.

A "width=N" argument will limit the number of revisions shown to N, defaulting to 5.

A "branch=BRANCHNAME" argument will limit the grid to revisions on branch BRANCHNAME.

`/tgrid`

The Transposed Grid is similar to the standard grid, but, as the name implies, transposes the grid: the revisions are listed down the left side of the page, and the build hosts are listed across the top. It accepts the same query arguments. The exception being that instead of "width" the argument is named "length."

`/console`

EXPERIMENTAL: This provides a developer-oriented display of the the last changes and how they affected the builders.

It allows a developer to quickly see the status of each builder for the first build including his or her change. A green box means that the change succeeded for all the steps for a given builder. A red box means that the changed introduced a new regression on a builder. An orange box means that at least one of the test failed, but it was also failing in the previous build, so it is not possible to see if there was any regressions from this change. Finally a yellow box means that the test is in progress.

By adding one or more "builder=" query arguments, the Console view is restricted to only showing information about the given Builders. By adding one or more "branch=" query arguments, the display is restricted to showing information about the given branches. In addition, adding one or more "category=" query arguments to the URL will limit the display to Builders that were defined with one of the given categories.

By adding one or more "name=" query arguments, the console view is restricted to only showing changes made by the given users.

NOTE: To use this page, your buildbot.css file in public_html must be the one found in buildbot/status/web/extended.css.

The console view is still in development. At this moment it supports only the source control managers that have an integer based revision id, like svn. It also has some issues with displaying multiple braches at the same time. If you do have multiple branches, you should use the "branch=" query argument.

`/rss`

This provides a rss feed summarizing all failed builds. The same query-arguments used by 'waterfall' can be added to filter the feed output.

`/atom`

This provides an atom feed summarizing all failed builds. The same query-arguments used by 'waterfall' can be added to filter the feed output.

`/buildstatus?builder=$BUILDERNAME&number=$BUILDNUM`

This displays a waterfall-like chronologically-oriented view of all the steps for a given build number on a given builder.

`/builders/$BUILDERNAME`

This describes the given Builder, and provides buttons to force a build. A `numbuilds=` argument will control how many build lines are displayed (5 by default).

`/builders/$BUILDERNAME/builds/$BUILDNUM`

This describes a specific Build.

`/builders/$BUILDERNAME/builds/$BUILDNUM/steps/$STEPNAME`

This describes a specific BuildStep.

`/builders/$BUILDERNAME/builds/$BUILDNUM/steps/$STEPNAME/logs/$LOGNAME`

This provides an HTML representation of a specific logfile.

`/builders/$BUILDERNAME/builds/$BUILDNUM/steps/$STEPNAME/logs/$LOGNAME/text`
: This returns the logfile as plain text, without any HTML coloring markup. It also removes the "headers", which are the lines that describe what command was run and what the environment variable settings were like. This maybe be useful for saving to disk and feeding to tools like 'grep'.

`/changes`
: This provides a brief description of the ChangeSource in use (see Section 5.1 [Change Sources], page 50).

`/changes/NN`
: This shows detailed information about the numbered Change: who was the author, what files were changed, what revision number was represented, etc.

`/buildslaves`
: This summarizes each BuildSlave, including which Builders are configured to use it, whether the buildslave is currently connected or not, and host information retrieved from the buildslave itself.

`/one_line_per_build`
: This page shows one line of text for each build, merging information from all Builders[2]. Each line specifies the name of the Builder, the number of the Build, what revision it used, and a summary of the results. Successful builds are in green, while failing builds are in red. The date and time of the build are added to the right-hand edge of the line. The lines are ordered by build finish timestamp.

  One or more `builder=` or `branch=` arguments can be used to restrict the list. In addition, a `numbuilds=` argument will control how many lines are displayed (20 by default).

`/one_box_per_builder`
: This page shows a small table, with one box for each Builder, containing the results of the most recent Build. It does not show the individual steps, or the current status. This is a simple summary of buildbot status: if this page is green, then all tests are passing.

  As with `/one_line_per_build`, this page will also honor `builder=` and `branch=` arguments.

`/about`
: This page gives a brief summary of the Buildbot itself: software version, versions of some libraries that the Buildbot depends upon, etc. It also contains a link to the buildbot.net home page.

`/slave_status_timeline`
: (note: this page has not yet been implemented)

  This provides a chronological display of configuration and operational events: master startup/shutdown, slave connect/disconnect, and config-file changes.

[2] Apparently this is the same way http://buildd.debian.org displays build status

When a config-file reload is abandoned because of an error in the config file, the error is displayed on this page.

This page does not show any builds.

`/last_build/$BUILDERNAME/status.png`
: This returns a PNG image that describes the results of the most recent build, which can be referenced in an IMG tag by other pages, perhaps from a completely different site. Use it as you would a webcounter.

There are also a set of web-status resources that are intended for use by other programs, rather than humans.

`/xmlrpc`
: This runs an XML-RPC server which can be used to query status information about various builds. See Section 7.1.4 [XMLRPC server], page 111 for more details.

7.1.4 XMLRPC server

When using WebStatus, the buildbot runs an XML-RPC server at '`/xmlrpc`' that can be used by other programs to query build status. The following table lists the methods that can be invoked using this interface.

`getAllBuildsInInterval(start, stop)`
: Return a list of builds that have completed after the 'start' timestamp and before the 'stop' timestamp. This looks at all Builders.

The timestamps are integers, interpreted as standard unix timestamps (seconds since epoch).

Each Build is returned as a tuple in the form: `(buildername, buildnumber, build_end, branchname, revision, results, text)`

The buildnumber is an integer. 'build_end' is an integer (seconds since epoch) specifying when the build finished.

The branchname is a string, which may be an empty string to indicate None (i.e. the default branch). The revision is a string whose meaning is specific to the VC system in use, and comes from the 'got_revision' build property. The results are expressed as a string, one of ('success', 'warnings', 'failure', 'exception'). The text is a list of short strings that ought to be joined by spaces and include slightly more data about the results of the build.

`getBuild(builder_name, build_number)`
: Return information about a specific build.

This returns a dictionary (aka "struct" in XMLRPC terms) with complete information about the build. It does not include the contents of the log files, but it has just about everything else.

7.1.5 HTML Waterfall

The `Waterfall` status target, deprecated as of 0.7.6, is a subset of the regular `WebStatus` resource (see Section 7.1 [WebStatus], page 105). This section (and the `Waterfall` class itself) will be removed from a future release.

```
from buildbot.status import html
w = html.WebStatus(http_port=8080)
c['status'].append(w)
```

7.2 MailNotifier

The buildbot can also send email when builds finish. The most common use of this is to tell developers when their change has caused the build to fail. It is also quite common to send a message to a mailing list (usually named "builds" or similar) about every build.

The `MailNotifier` status target is used to accomplish this. You configure it by specifying who mail should be sent to, under what circumstances mail should be sent, and how to deliver the mail. It can be configured to only send out mail for certain builders, and only send messages when the build fails, or when the builder transitions from success to failure. It can also be configured to include various build logs in each message.

By default, the message will be sent to the Interested Users list (see Section 3.6.1 [Doing Things With Users], page 25), which includes all developers who made changes in the build. You can add additional recipients with the extraRecipients argument.

Each MailNotifier sends mail to a single set of recipients. To send different kinds of mail to different recipients, use multiple MailNotifiers.

The following simple example will send an email upon the completion of each build, to just those developers whose Changes were included in the build. The email contains a description of the Build, its results, and URLs where more information can be obtained.

```
from buildbot.status.mail import MailNotifier
mn = MailNotifier(fromaddr="buildbot@example.org", lookup="example.org")
c['status'].append(mn)
```

To get a simple one-message-per-build (say, for a mailing list), use the following form instead. This form does not send mail to individual developers (and thus does not need the `lookup=` argument, explained below), instead it only ever sends mail to the "extra recipients" named in the arguments:

```
mn = MailNotifier(fromaddr="buildbot@example.org",
                  sendToInterestedUsers=False,
                  extraRecipients=['listaddr@example.org'])
```

In some cases it is desirable to have different information then what is provided in a standard MailNotifier message. For this purpose MailNotifier provides the argument `messageFormatter` (a function) which allows for the creation of messages with unique content.

For example, if only short emails are desired (e.g., for delivery to phones)

```
from buildbot.status.builder import Results
def messageFormatter(mode, name, build, results, master_status):
    result = Results[results]

    text = list()
    text.append("STATUS: %s" % result.title())
    return {
        'body' : "\n".join(text),
```

```
        'type' : 'plain'
    }

mn = MailNotifier(fromaddr="buildbot@example.org",
                  sendToInterestedUsers=False,
                  mode='problem',
                  extraRecipients=['listaddr@example.org'],
                  messageFormatter=messageFormatter)
```

Another example of a function delivering a customized html email containing the last 80 lines of logs of a failing build step is given below:

```
from buildbot.status.builder import Results

def message_formatter(mode, name, build, results, master_status):
    """Provide a customized message to BuildBot's MailNotifier.

    The last 80 lines of the log are provided as well as the changes
    relevant to the build.  Message content is formatted as html.
    """
    result = Results[results]

    limit_lines = 80
    text = list()
    text.append('<h4>Build status: %s</h4>' % result.upper())
    text.append("Buildslave for this Build: <b>%s</b>" % build.getSlavename())
    text.append('<br>')
    if master_status.getURLForThing(build):
        text.append('Complete logs for all build steps: <a href="%s">%s</a>'
                    % (master_status.getURLForThing(build),
                       master_status.getURLForThing(build))
                    )
    text.append('<br>')
    text.append("Build Reason: %s" % build.getReason())
    text.append('<br>')

    source = ""
    ss = build.getSourceStamp()
    if ss.branch:
        source += "[branch %s] " % ss.branch
    if ss.revision:
        source +=  ss.revision
    else:
        source += "HEAD"
    if ss.patch:
        source += " (plus patch)"
    text.append("Build Source Stamp: <b>%s</b>" % source)
    text.append('<br>')
```

```
        text.append("Blamelist: %s" % ",".join(build.getResponsibleUsers()))
        if ss.changes:
            text.append('<h4>Recent Changes:</h4>')
            text.extend([c.asHTML() for c in ss.changes])

        logs = list()
        for log in build.getLogs():
            log_name = "%s.%s" % (log.getStep().getName(), log.getName())
            log_status, dummy = log.getStep().getResults()
            log_body = log.getText().splitlines() # Note: can be VERY LARGE
            log_url = '%s/steps/%s/logs/%s' % (master_status.getURLForThing(build),
                                               log.getStep().getName(),
                                               log.getName())
            logs.append((log_name, log_url, log_body, log_status))

        name, url, content, logstatus = logs[-1]

        text.append('<i>Detailed log of last build step:</i> <a href="%s">%s</a>'
                    % (url, url))
        text.append('<br>')
        text.append('<h4>Last %d lines of "%s":</h4>' % (limit_lines, name))
        text.append('<p>')
        text.append('<br>'.join([line for line in
                                 content[len(content)-limit_lines:]]))
        text.append('</p>')
        text.append('<br><br>')
        text.append('<b>-The BuildBot</b>')
        return {
            'body': "\n".join(text),
            'type': 'html'
            }

    mn = MailNotifier(fromaddr="buildbot@example.org",
                      sendToInterestedUsers=False,
                      mode='failing',
                      extraRecipients=['listaddr@example.org'],
                      messageFormatter=message_formatter)
```

MailNotifier arguments

`fromaddr` The email address to be used in the 'From' header.

`sendToInterestedUsers`
(boolean). If True (the default), send mail to all of the Interested Users. If False, only send mail to the extraRecipients list.

`extraRecipients`
: (tuple of strings). A list of email addresses to which messages should be sent (in addition to the InterestedUsers list, which includes any developers who made Changes that went into this build). It is a good idea to create a small mailing list and deliver to that, then let subscribers come and go as they please.

`subject`
: (string). A string to be used as the subject line of the message. `%(builder)s` will be replaced with the name of the builder which provoked the message.

`mode`
: (string). Default to 'all'. One of:

  `all` Send mail about all builds, bothpassing and failing

  `failing` Only send mail about builds which fail

  `problem` Only send mail about a build which failed when the previous build has passed. If your builds usually pass, then this will only send mail when a problem occurs.

`builders`
: (list of strings). A list of builder names for which mail should be sent. Defaults to None (send mail for all builds). Use either builders or categories, but not both.

`categories`
: (list of strings). A list of category names to serve status information for. Defaults to None (all categories). Use either builders or categories, but not both.

`addLogs`
: (boolean). If True, include all build logs as attachments to the messages. These can be quite large. This can also be set to a list of log names, to send a subset of the logs. Defaults to False.

`addPatch`
: (boolean). If True, include the patch content if a patch was present. Patches are usually used on a Try server. Defaults to True.

`relayhost`
: (string). The host to which the outbound SMTP connection should be made. Defaults to 'localhost'

`lookup`
: (implementor of `IEmailLookup`). Object which provides IEmailLookup, which is responsible for mapping User names (which come from the VC system) into valid email addresses. If not provided, the notifier will only be able to send mail to the addresses in the extraRecipients list. Most of the time you can use a simple Domain instance. As a shortcut, you can pass as string: this will be treated as if you had provided Domain(str). For example, lookup='twistedmatrix.com' will allow mail to be sent to all developers whose SVN usernames match their twistedmatrix.com account names. See buildbot/status/mail.py for more details.

`messageFormatter`
: This is a optional function that can be used to generate a custom mail message. A `messageFormatter` function takes the mail mode (`mode`), builder name (`name`), the build status (`build`), the result code (`results`), and the BuildMaster status (`master_status`). It returns a dictionary. The `body` key gives a string that is the complete text of the message. The `type` key is the message

type ('plain' or 'html'). The 'html' type should be used when generating an HTML message. The `subject` key is optional, but gives the subject for the email.

`extraHeaders`
(dictionary) A dictionary containing key/value pairs of extra headers to add to sent e-mails. Both the keys and the values may be a WithProperties instance.

As a help to those writing `messageFormatter` functions, the following table describes how to get some useful pieces of information from the various status objects:

•Name of the builder that generated this event
`name`

•Name of the project
`master_status.getProjectName()`

•MailNotifier mode
`mode` (one of `all`, `failing`, `problem`, `change`, `passing`)

•Builder result as a string
```
from buildbot.status.builder import Results
result_str = Results[results]
# one of 'success', 'warnings', 'failure', 'skipped', or 'exception'
```

•URL to build page
`master_status.getURLForThing(build)`

•URL to buildbot main page.
`master_status.getBuildbotURL()`

•Build text
`build.getText()`

•Mapping of property names to values
`build.getProperties()` (a `Properties` instance)

•Slave name
`build.getSlavename()`

•Build reason (from a forced build)
`build.getReason()`

•List of responsible users
`build.getResponsibleUsers()`

•Source information (only valid if ss is not None)
```
ss = build.getSourceStamp()
if ss:
    branch = ss.branch
    revision = ss.revision
    patch = ss.patch
    changes = ss.changes # list
```
A change object has the following useful information:

`who` (str) who made this change

`revision` (str) what VC revision is this change

`branch` (str) on what branch did this change occur

`when` (str) when did this change occur

`files` (list of str) what files were affected in this change

`comments` (str) comments reguarding the change.

The `Change` methods asText and asHTML return a list of strings with the above information formatted.

•Log information

```
logs = list()
for log in build.getLogs():
    log_name = "%s.%s" % (log.getStep().getName(), log.getName())
    log_status, dummy = log.getStep().getResults()
    log_body = log.getText().splitlines() # Note: can be VERY LARGE
    log_url = '%s/steps/%s/logs/%s' % (master_status.getURLForThing(build
                                       log.getStep().getName(),
                                       log.getName())
    logs.append((log_name, log_url, log_body, log_status))
```

7.3 IRC Bot

The `buildbot.status.words.IRC` status target creates an IRC bot which will attach to certain channels and be available for status queries. It can also be asked to announce builds as they occur, or be told to shut up.

```
from buildbot.status import words
irc = words.IRC("irc.example.org", "botnickname",
                channels=["channel1", "channel2"],
                password="mysecretpassword",
                notify_events={
                  'exception': 1,
                  'successToFailure': 1,
                  'failureToSuccess': 1,
                  })
c['status'].append(irc)
```

Take a look at the docstring for `words.IRC` for more details on configuring this service. The `password` argument, if provided, will be sent to Nickserv to claim the nickname: some IRC servers will not allow clients to send private messages until they have logged in with a password.

To use the service, you address messages at the buildbot, either normally (`botnickname: status`) or with private messages (`/msg botnickname status`). The buildbot will respond in kind.

Some of the commands currently available:

`list builders`

Emit a list of all configured builders

`status BUILDER`
: Announce the status of a specific Builder: what it is doing right now.

`status all`
: Announce the status of all Builders

`watch BUILDER`
: If the given Builder is currently running, wait until the Build is finished and then announce the results.

`last BUILDER`
: Return the results of the last build to run on the given Builder.

`join CHANNEL`
: Join the given IRC channel

`leave CHANNEL`
: Leave the given IRC channel

`notify on|off|list EVENT`
: Report events relating to builds. If the command is issued as a private message, then the report will be sent back as a private message to the user who issued the command. Otherwise, the report will be sent to the channel. Available events to be notified are:

  `started` A build has started

  `finished` A build has finished

  `success` A build finished successfully

  `failed` A build failed

  `exception`
  : A build generated and exception

  `xToY` The previous build was x, but this one is Y, where x and Y are each one of success, warnings, failure, exception (except Y is capitalized). For example: successToFailure will notify if the previous build was successful, but this one failed

`help COMMAND`
: Describe a command. Use `help commands` to get a list of known commands.

`source` Announce the URL of the Buildbot's home page.

`version` Announce the version of this Buildbot.

Additionally, the config file may specify default notification options as shown in the example earlier.

If the `allowForce=True` option was used, some addtional commands will be available:

`force build BUILDER REASON`
: Tell the given Builder to start a build of the latest code. The user requesting the build and REASON are recorded in the Build status. The buildbot will announce the build's status when it finishes.

`stop build BUILDER REASON`

Terminate any running build in the given Builder. REASON will be added to the build status to explain why it was stopped. You might use this if you committed a bug, corrected it right away, and don't want to wait for the first build (which is destined to fail) to complete before starting the second (hopefully fixed) build.

7.4 PBListener

```
import buildbot.status.client
pbl = buildbot.status.client.PBListener(port=int, user=str,
                                        passwd=str)
c['status'].append(pbl)
```

This sets up a PB listener on the given TCP port, to which a PB-based status client can connect and retrieve status information. `buildbot statusgui` (see Section 8.2.2 [statusgui], page 121) is an example of such a status client. The `port` argument can also be a strports specification string.

7.5 Writing New Status Plugins

TODO: this needs a lot more examples

Each status plugin is an object which provides the `twisted.application.service.IService` interface, which creates a tree of Services with the buildmaster at the top [not strictly true]. The status plugins are all children of an object which implements `buildbot.interfaces.IStatus`, the main status object. From this object, the plugin can retrieve anything it wants about current and past builds. It can also subscribe to hear about new and upcoming builds.

Status plugins which only react to human queries (like the Waterfall display) never need to subscribe to anything: they are idle until someone asks a question, then wake up and extract the information they need to answer it, then they go back to sleep. Plugins which need to act spontaneously when builds complete (like the MailNotifier plugin) need to subscribe to hear about new builds.

If the status plugin needs to run network services (like the HTTP server used by the Waterfall plugin), they can be attached as Service children of the plugin itself, using the `IServiceCollection` interface.

8 Command-line tool

The `buildbot` command-line tool can be used to start or stop a buildmaster or buildbot, and to interact with a running buildmaster. Some of its subcommands are intended for buildmaster admins, while some are for developers who are editing the code that the buildbot is monitoring.

8.1 Administrator Tools

The following `buildbot` sub-commands are intended for buildmaster administrators:

create-master

This creates a new directory and populates it with files that allow it to be used as a buildmaster's base directory.

```
buildbot create-master BASEDIR
```

create-slave

This creates a new directory and populates it with files that let it be used as a buildslave's base directory. You must provide several arguments, which are used to create the initial '`buildbot.tac`' file.

```
buildbot create-slave BASEDIR MASTERHOST:PORT SLAVENAME PASSWORD
```

start

This starts a buildmaster or buildslave which was already created in the given base directory. The daemon is launched in the background, with events logged to a file named '`twistd.log`'.

```
buildbot start BASEDIR
```

stop

This terminates the daemon (either buildmaster or buildslave) running in the given directory.

```
buildbot stop BASEDIR
```

sighup

This sends a SIGHUP to the buildmaster running in the given directory, which causes it to re-read its '`master.cfg`' file.

```
buildbot sighup BASEDIR
```

8.2 Developer Tools

These tools are provided for use by the developers who are working on the code that the buildbot is monitoring.

8.2.1 statuslog

```
buildbot statuslog --master MASTERHOST:PORT
```

This command starts a simple text-based status client, one which just prints out a new line each time an event occurs on the buildmaster.

The '`--master`' option provides the location of the `buildbot.status.client.PBListener` status port, used to deliver build information to realtime status clients. The option is always in the form of a string, with hostname and port number separated by a colon (`HOSTNAME:PORTNUM`). Note that this port is *not* the same as the slaveport (although a future version may allow the same port number to be used for both purposes). If you get an error message to the effect of "Failure: twisted.cred.error.UnauthorizedLogin:", this may indicate that you are connecting to the slaveport rather than a `PBListener` port.

The '`--master`' option can also be provided by the `masterstatus` name in '`.buildbot/options`' (see Section 8.4 [.buildbot config directory], page 129).

8.2.2 statusgui

If you have set up a PBListener (see Section 7.4 [PBListener], page 119), you will be able to monitor your Buildbot using a simple Gtk+ application invoked with the `buildbot statusgui` command:

```
buildbot statusgui --master MASTERHOST:PORT
```

This command starts a simple Gtk+-based status client, which contains a few boxes for each Builder that change color as events occur. It uses the same '`--master`' argument and `masterstatus` option as the `buildbot statuslog` command (see Section 8.2.1 [statuslog], page 121).

8.2.3 try

This lets a developer to ask the question "What would happen if I committed this patch right now?". It runs the unit test suite (across multiple build platforms) on the developer's current code, allowing them to make sure they will not break the tree when they finally commit their changes.

The `buildbot try` command is meant to be run from within a developer's local tree, and starts by figuring out the base revision of that tree (what revision was current the last time the tree was updated), and a patch that can be applied to that revision of the tree to make it match the developer's copy. This (revision, patch) pair is then sent to the buildmaster, which runs a build with that SourceStamp. If you want, the tool will emit status messages as the builds run, and will not terminate until the first failure has been detected (or the last success).

There is an alternate form which accepts a pre-made patch file (typically the output of a command like 'svn diff'). This "–diff" form does not require a local tree to run from. See See Section 8.2.3.1 [try –diff], page 126.

For this command to work, several pieces must be in place:

TryScheduler

The buildmaster must have a `scheduler.Try` instance in the config file's `c['schedulers']` list. This lets the administrator control who may initiate these "trial" builds, which branches are eligible for trial builds, and which Builders should be used for them.

The `TryScheduler` has various means to accept build requests: all of them enforce more security than the usual buildmaster ports do. Any source code being built can be used to compromise the buildslave accounts, but in general that code must be checked out from the VC repository first, so only people with commit privileges can get control of the buildslaves. The usual force-build control channels can waste buildslave time but do not allow arbitrary commands to be executed by people who don't have those commit privileges. However, the source code patch that is provided with the trial build does not have to go through the VC system first, so it is important to make sure these builds cannot be abused by a non-committer to acquire as much control over the buildslaves as a committer has. Ideally, only developers who have commit access to the VC repository would be able to start trial builds, but unfortunately the buildmaster does not, in general, have access to VC system's user list.

As a result, the `TryScheduler` requires a bit more configuration. There are currently two ways to set this up:

jobdir (ssh)

This approach creates a command queue directory, called the "jobdir", in the buildmaster's working directory. The buildmaster admin sets the ownership and permissions of this directory to only grant write access to the desired set of developers, all of whom must have accounts on the machine. The `buildbot try` command creates a special file containing the source stamp information and drops it in the jobdir, just like a standard maildir. When the buildmaster notices the new file, it unpacks the information inside and starts the builds.

The config file entries used by 'buildbot try' either specify a local queuedir (for which write and mv are used) or a remote one (using scp and ssh).

The advantage of this scheme is that it is quite secure, the disadvantage is that it requires fiddling outside the buildmaster config (to set the permissions on the jobdir correctly). If the buildmaster machine happens to also house the VC repository, then it can be fairly easy to keep the VC userlist in sync with the trial-build userlist. If they are on different machines, this will be much more of a hassle. It may also involve granting developer accounts on a machine that would not otherwise require them.

To implement this, the buildslave invokes 'ssh -l username host buildbot tryserver ARGS', passing the patch contents over stdin. The arguments must include the inlet directory and the revision information.

user+password (PB)

In this approach, each developer gets a username/password pair, which are all listed in the buildmaster's configuration file. When the developer runs `buildbot try`, their machine connects to the buildmaster via PB and authenticates themselves using that username and password, then sends a PB command to start the trial build.

The advantage of this scheme is that the entire configuration is performed inside the buildmaster's config file. The disadvantages are that it is less secure (while the "cred" authentication system does not expose the password in plaintext over the wire, it does not offer most of the other security properties that SSH

does). In addition, the buildmaster admin is responsible for maintaining the username/password list, adding and deleting entries as developers come and go.

For example, to set up the "jobdir" style of trial build, using a command queue directory of '`MASTERDIR/jobdir`' (and assuming that all your project developers were members of the `developers` unix group), you would first create that directory (with `mkdir MASTERDIR/jobdir MASTERDIR/jobdir/new MASTERDIR/jobdir/cur MASTERDIR/jobdir/tmp; chgrp developers MASTERDIR/jobdir MASTERDIR/jobdir/*; chmod g+rwx,o-rwx MASTERDIR/jobdir MASTERDIR/jobdir/*`), and then use the following scheduler in the buildmaster's config file:

```
from buildbot.scheduler import Try_Jobdir
s = Try_Jobdir("try1", ["full-linux", "full-netbsd", "full-OSX"],
               jobdir="jobdir")
c['schedulers'] = [s]
```

Note that you must create the jobdir before telling the buildmaster to use this configuration, otherwise you will get an error. Also remember that the buildmaster must be able to read and write to the jobdir as well. Be sure to watch the '`twistd.log`' file (see Section 2.7 [Logfiles], page 14) as you start using the jobdir, to make sure the buildmaster is happy with it.

To use the username/password form of authentication, create a `Try_Userpass` instance instead. It takes the same `builderNames` argument as the `Try_Jobdir` form, but accepts an addtional `port` argument (to specify the TCP port to listen on) and a `userpass` list of username/password pairs to accept. Remember to use good passwords for this: the security of the buildslave accounts depends upon it:

```
from buildbot.scheduler import Try_Userpass
s = Try_Userpass("try2", ["full-linux", "full-netbsd", "full-OSX"],
                 port=8031, userpass=[("alice","pw1"), ("bob", "pw2")] )
c['schedulers'] = [s]
```

Like most places in the buildbot, the `port` argument takes a strports specification. See `twisted.application.strports` for details.

locating the master

The `try` command needs to be told how to connect to the `TryScheduler`, and must know which of the authentication approaches described above is in use by the buildmaster. You specify the approach by using '`--connect=ssh`' or '`--connect=pb`' (or `try_connect = 'ssh'` or `try_connect = 'pb'` in '`.buildbot/options`').

For the PB approach, the command must be given a '`--master`' argument (in the form HOST:PORT) that points to TCP port that you picked in the `Try_Userpass` scheduler. It also takes a '`--username`' and '`--passwd`' pair of arguments that match one of the entries in the buildmaster's `userpass` list. These arguments can also be provided as `try_master`, `try_username`, and `try_password` entries in the '`.buildbot/options`' file.

For the SSH approach, the command must be given '`--tryhost`', '`--username`', and optionally '`--password`' (TODO: really?) to get to the buildmaster host. It must also be given '`--trydir`', which points to the inlet directory configured above. The trydir can be relative to the user's home directory, but most of the time you will use an explicit path like

‘~buildbot/project/trydir’. These arguments can be provided in ‘.buildbot/options’ as try_host, try_username, try_password, and try_dir.

In addition, the SSH approach needs to connect to a PBListener status port, so it can retrieve and report the results of the build (the PB approach uses the existing connection to retrieve status information, so this step is not necessary). This requires a ‘--master’ argument, or a masterstatus entry in ‘.buildbot/options’, in the form of a HOSTNAME:PORT string.

choosing the Builders

A trial build is performed on multiple Builders at the same time, and the developer gets to choose which Builders are used (limited to a set selected by the buildmaster admin with the TryScheduler’s builderNames= argument). The set you choose will depend upon what your goals are: if you are concerned about cross-platform compatibility, you should use multiple Builders, one from each platform of interest. You might use just one builder if that platform has libraries or other facilities that allow better test coverage than what you can accomplish on your own machine, or faster test runs.

The set of Builders to use can be specified with multiple ‘--builder’ arguments on the command line. It can also be specified with a single try_builders option in ‘.buildbot/options’ that uses a list of strings to specify all the Builder names:

```
try_builders = ["full-OSX", "full-win32", "full-linux"]
```

specifying the VC system

The try command also needs to know how to take the developer’s current tree and extract the (revision, patch) source-stamp pair. Each VC system uses a different process, so you start by telling the try command which VC system you are using, with an argument like ‘--vc=cvs’ or ‘--vc=tla’. This can also be provided as try_vc in ‘.buildbot/options’.

The following names are recognized: cvs svn baz tla hg darcs

finding the top of the tree

Some VC systems (notably CVS and SVN) track each directory more-or-less independently, which means the try command needs to move up to the top of the project tree before it will be able to construct a proper full-tree patch. To accomplish this, the try command will crawl up through the parent directories until it finds a marker file. The default name for this marker file is ‘.buildbot-top’, so when you are using CVS or SVN you should touch .buildbot-top from the top of your tree before running buildbot try. Alternatively, you can use a filename like ‘ChangeLog’ or ‘README’, since many projects put one of these files in their top-most directory (and nowhere else). To set this filename, use ‘--try-topfile=ChangeLog’, or set it in the options file with try_topfile = ’ChangeLog’.

You can also manually set the top of the tree with ‘--try-topdir=~/trees/mytree’, or try_topdir = ’~/trees/mytree’. If you use try_topdir, in a ‘.buildbot/options’ file, you will need a separate options file for each tree you use, so it may be more convenient to use the try_topfile approach instead.

Other VC systems which work on full projects instead of individual directories (tla, baz, darcs, monotone, mercurial, git) do not require `try` to know the top directory, so the '`--try-topfile`' and '`--try-topdir`' arguments will be ignored.

If the `try` command cannot find the top directory, it will abort with an error message.

determining the branch name

Some VC systems record the branch information in a way that "try" can locate it, in particular Arch (both `tla` and `baz`). For the others, if you are using something other than the default branch, you will have to tell the buildbot which branch your tree is using. You can do this with either the '`--branch`' argument, or a '`try_branch`' entry in the '`.buildbot/options`' file.

determining the revision and patch

Each VC system has a separate approach for determining the tree's base revision and computing a patch.

`CVS`

: `try` pretends that the tree is up to date. It converts the current time into a `-D` time specification, uses it as the base revision, and computes the diff between the upstream tree as of that point in time versus the current contents. This works, more or less, but requires that the local clock be in reasonably good sync with the repository.

`SVN`

: `try` does a `svn status -u` to find the latest repository revision number (emitted on the last line in the "Status against revision: NN" message). It then performs an `svn diff -rNN` to find out how your tree differs from the repository version, and sends the resulting patch to the buildmaster. If your tree is not up to date, this will result in the "try" tree being created with the latest revision, then *backwards* patches applied to bring it "back" to the version you actually checked out (plus your actual code changes), but this will still result in the correct tree being used for the build.

`baz`

: `try` does a `baz tree-id` to determine the fully-qualified version and patch identifier for the tree (ARCHIVE/VERSION–patch-NN), and uses the VERSION–patch-NN component as the base revision. It then does a `baz diff` to obtain the patch.

`tla`

: `try` does a `tla tree-version` to get the fully-qualified version identifier (ARCHIVE/VERSION), then takes the first line of `tla logs --reverse` to figure out the base revision. Then it does `tla changes --diffs` to obtain the patch.

`Darcs`

: `darcs changes --context` emits a text file that contains a list of all patches back to and including the last tag was made. This text file (plus the location of a repository that contains all these patches) is sufficient to re-create the tree. Therefore the contents of this "context" file *are* the revision stamp for a Darcs-controlled source tree.

So `try` does a `darcs changes --context` to determine what your tree's base revision is, and then does a `darcs diff -u` to compute the patch relative to that revision.

`Mercurial`

`hg identify` emits a short revision ID (basically a truncated SHA1 hash of the current revision's contents), which is used as the base revision. `hg diff` then provides the patch relative to that revision. For `try` to work, your working directory must only have patches that are available from the same remotely-available repository that the build process' `source.Mercurial` will use.

`Git` `git branch -v` lists all the branches available in the local repository along with the revision ID it points to and a short summary of the last commit. The line containing the currently checked out branch begins with '* ' (star and space) while all the others start with ' ' (two spaces). `try` scans for this line and extracts the branch name and revision from it. Then it generates a diff against the base revision.

`Monotone` `mtn automate get_base_revisions_id` emits the base revision that we use. `mtn diff` then provides the patch relative to that revision. For `try` to work, your working directory must only have patches that are available from the same remotely-available repository that the build process' `source.Mercurial` will use.

waiting for results

If you provide the '`--wait`' option (or `try_wait = True` in '`.buildbot/options`'), the `buildbot try` command will wait until your changes have either been proven good or bad before exiting. Unless you use the '`--quiet`' option (or `try_quiet=True`), it will emit a progress message every 60 seconds until the builds have completed.

8.2.3.1 try –diff

Sometimes you might have a patch from someone else that you want to submit to the buildbot. For example, a user may have created a patch to fix some specific bug and sent it to you by email. You've inspected the patch and suspect that it might do the job (and have at least confirmed that it doesn't do anything evil). Now you want to test it out.

One approach would be to check out a new local tree, apply the patch, run your local tests, then use "buildbot try" to run the tests on other platforms. An alternate approach is to use the `buildbot try --diff` form to have the buildbot test the patch without using a local tree.

This form takes a '`--diff`' argument which points to a file that contains the patch you want to apply. By default this patch will be applied to the TRUNK revision, but if you give the optional '`--baserev`' argument, a tree of the given revision will be used as a starting point instead of TRUNK.

You can also use `buildbot try --diff=-` to read the patch from stdin.

Each patch has a "patchlevel" associated with it. This indicates the number of slashes (and preceding pathnames) that should be stripped before applying the diff. This exactly

corresponds to the '`-p`' or '`--strip`' argument to the `patch` utility. By default `buildbot try --diff` uses a patchlevel of 0, but you can override this with the '`-p`' argument.

When you use '`--diff`', you do not need to use any of the other options that relate to a local tree, specifically '`--vc`', '`--try-topfile`', or '`--try-topdir`'. These options will be ignored. Of course you must still specify how to get to the buildmaster (with '`--connect`', '`--tryhost`', etc).

8.3 Other Tools

These tools are generally used by buildmaster administrators.

8.3.1 sendchange

This command is used to tell the buildmaster about source changes. It is intended to be used from within a commit script, installed on the VC server. It requires that you have a PBChangeSource (see Section 5.5 [PBChangeSource], page 56) running in the buildmaster (by being set in `c['change_source']`).

```
buildbot sendchange --master MASTERHOST:PORT --username USER FILENAMES..
```

The `master` and `username` arguments can also be given in the options file (see Section 8.4 [.buildbot config directory], page 129). There are other (optional) arguments which can influence the `Change` that gets submitted:

`--branch` (or option `branch`) This provides the (string) branch specifier. If omitted, it defaults to None, indicating the "default branch". All files included in this Change must be on the same branch.

`--category`
(or option `category`) This provides the (string) category specifier. If omitted, it defaults to None, indicating "no category". The category property is used by Schedulers to filter what changes they listen to.

`--revision_number`
This provides a (numeric) revision number for the change, used for VC systems that use numeric transaction numbers (like Subversion).

`--revision`
This provides a (string) revision specifier, for VC systems that use strings (Arch would use something like patch-42 etc).

`--revision_file`
This provides a filename which will be opened and the contents used as the revision specifier. This is specifically for Darcs, which uses the output of `darcs changes --context` as a revision specifier. This context file can be a couple of kilobytes long, spanning a couple lines per patch, and would be a hassle to pass as a command-line argument.

`--property`
This parameter is used to set a property on the Change generated by sendchange. Properties are specified as a name:value pair, separated by a colon. You may specify many properties by passing this parameter multiple times.

`--comments`
: This provides the change comments as a single argument. You may want to use '`--logfile`' instead.

`--logfile`
: This instructs the tool to read the change comments from the given file. If you use `-` as the filename, the tool will read the change comments from stdin.

8.3.2 debugclient

```
buildbot debugclient --master MASTERHOST:PORT --passwd DEBUGPW
```

This launches a small Gtk+/Glade-based debug tool, connecting to the buildmaster's "debug port". This debug port shares the same port number as the slaveport (see Section 4.8 [Setting the slaveport], page 39), but the `debugPort` is only enabled if you set a debug password in the buildmaster's config file (see Section 4.15 [Debug options], page 48). The '`--passwd`' option must match the `c['debugPassword']` value.

'`--master`' can also be provided in '`.debug/options`' by the `master` key. '`--passwd`' can be provided by the `debugPassword` key. See Section 8.4 [.buildbot config directory], page 129.

The `Connect` button must be pressed before any of the other buttons will be active. This establishes the connection to the buildmaster. The other sections of the tool are as follows:

`Reload .cfg`
: Forces the buildmaster to reload its '`master.cfg`' file. This is equivalent to sending a SIGHUP to the buildmaster, but can be done remotely through the debug port. Note that it is a good idea to be watching the buildmaster's '`twistd.log`' as you reload the config file, as any errors which are detected in the config file will be announced there.

`Rebuild .py`
: (not yet implemented). The idea here is to use Twisted's "rebuild" facilities to replace the buildmaster's running code with a new version. Even if this worked, it would only be used by buildbot developers.

`poke IRC` This locates a `words.IRC` status target and causes it to emit a message on all the channels to which it is currently connected. This was used to debug a problem in which the buildmaster lost the connection to the IRC server and did not attempt to reconnect.

`Commit` This allows you to inject a Change, just as if a real one had been delivered by whatever VC hook you are using. You can set the name of the committed file and the name of the user who is doing the commit. Optionally, you can also set a revision for the change. If the revision you provide looks like a number, it will be sent as an integer, otherwise it will be sent as a string.

`Force Build`
: This lets you force a Builder (selected by name) to start a build of the current source tree.

`Currently`
: (obsolete). This was used to manually set the status of the given Builder, but the status-assignment code was changed in an incompatible way and these buttons are no longer meaningful.

8.4 .buildbot config directory

Many of the `buildbot` tools must be told how to contact the buildmaster that they interact with. This specification can be provided as a command-line argument, but most of the time it will be easier to set them in an "options" file. The `buildbot` command will look for a special directory named '`.buildbot`', starting from the current directory (where the command was run) and crawling upwards, eventually looking in the user's home directory. It will look for a file named '`options`' in this directory, and will evaluate it as a python script, looking for certain names to be set. You can just put simple `name = 'value'` pairs in this file to set the options.

For a description of the names used in this file, please see the documentation for the individual `buildbot` sub-commands. The following is a brief sample of what this file's contents could be.

```
# for status-reading tools
masterstatus = 'buildbot.example.org:12345'
# for 'sendchange' or the debug port
master = 'buildbot.example.org:18990'
debugPassword = 'eiv7Po'
```

Note carefully that the names in the `options` file usually do not match the command-line option name.

`masterstatus`
: Equivalent to `--master` for Section 8.2.1 [statuslog], page 121 and Section 8.2.2 [statusgui], page 121, this gives the location of the `client.PBListener` status port.

`master`
: Equivalent to `--master` for Section 8.3.2 [debugclient], page 128 and Section 8.3.1 [sendchange], page 127. This option is used for two purposes. It is the location of the `debugPort` for `debugclient` and the location of the `pb.PBChangeSource` for `sendchange`. Generally these are the same port.

`debugPassword`
: Equivalent to `--passwd` for Section 8.3.2 [debugclient], page 128.

  XXX Must match the value of `c['debugPassword']`, used to protect the debug port, for the `debugclient` command.

`username`
: Equivalent to `--username` for the Section 8.3.1 [sendchange], page 127 command.

`branch`
: Equivalent to `--branch` for the Section 8.3.1 [sendchange], page 127 command.

`category`
: Equivalent to `--category` for the Section 8.3.1 [sendchange], page 127 command.

`try_connect`
: Equivalent to `--connect`, this specifies how the Section 8.2.3 [try], page 121 command should deliver its request to the buildmaster. The currently accepted values are "ssh" and "pb".

`try_builders`
: Equivalent to `--builders`, specifies which builders should be used for the Section 8.2.3 [try], page 121 build.

`try_vc`
: Equivalent to `--vc` for Section 8.2.3 [try], page 121, this specifies the version control system being used.

`try_branch`
: Equivlanent to `--branch`, this indicates that the current tree is on a non-trunk branch.

`try_topdir`
`try_topfile`
: Use `try_topdir`, equivalent to `--try-topdir`, to explicitly indicate the top of your working tree, or `try_topfile`, equivalent to `--try-topfile` to name a file that will only be found in that top-most directory.

`try_host`

`try_username`
`try_dir`
: When `try_connect` is "ssh", the command will use `try_host` for `--tryhost`, `try_username` for `--username`, and `try_dir` for `--trydir`. Apologies for the confusing presence and absence of 'try'.

`try_username`
`try_password`
`try_master`
: Similarly, when `try_connect` is "pb", the command will pay attention to `try_username` for `--username`, `try_password` for `--passwd`, and `try_master` for `--master`.

`try_wait`

`masterstatus`
: `try_wait` and `masterstatus` (equivalent to `--wait` and `master`, respectively) are used to ask the Section 8.2.3 [try], page 121 command to wait for the requested build to complete.

9 Resources

The Buildbot home page is `http://buildbot.net/`.

For configuration questions and general discussion, please use the `buildbot-devel` mailing list. The subscription instructions and archives are available at `http://lists.sourceforge.net/lists/listinfo/buildbot-devel`

The #buildbot channel on Freenode's IRC servers hosts development discussion, and often folks are available to answer questions there, as well.

Developer's Appendix

This appendix contains random notes about the implementation of the Buildbot, and is likely to only be of use to people intending to extend the Buildbot's internals.

The buildmaster consists of a tree of Service objects, which is shaped as follows:

```
BuildMaster
 ChangeMaster  (in .change_svc)
  [IChangeSource instances]
 [IScheduler instances]  (in .schedulers)
 BotMaster  (in .botmaster)
  [IBuildSlave instances]
 [IStatusTarget instances]  (in .statusTargets)
```

The BotMaster has a collection of Builder objects as values of its `.builders` dictionary.

Index of Useful Classes

This is a list of all user-visible classes. There are the ones that are useful in '`master.cfg`', the buildmaster's configuration file. Classes that are not listed here are generally internal things that admins are unlikely to have much use for.

Change Sources

Schedulers and Locks

Build Factories

Build Steps

Status Targets

Index of master.cfg keys

This is a list of all of the significant keys in master.cfg . Recall that master.cfg is effectively a small python program with exactly one responsibility: create a dictionary named `BuildmasterConfig`. The keys of this dictionary are listed here. The beginning of the master.cfg file typically starts with something like:

```
BuildmasterConfig = c = {}
```

Therefore a config key of `change_source` will usually appear in master.cfg as `c['change_source']`.

Index