The `build2` Build Bot

Revision `0.6`, August 2017
This revision of the document describes the `build2` build bot `0.6.x` series.

Table of Contents

Preface

This document describes `bbot`, the `build2` build bot.

1 Introduction

2 Architecture

The `bbot` architecture includes several layers for security and manageability. At the top we have a `bbot` running in the *controller* mode. The controller monitors various *build sources* for *build tasks*. For example, a controller may poll a `brep` instances for any new packages to built as well as monitor a `git` repository for any new commits to test. There can be several layers of controllers with `brep` being just a special kind. A machine running a `bbot` instance in the controller mode is called a *controller host*.

Below the controllers we have a `bbot` running in the *agent* mode normally on Build OS. The agent polls its controllers for *build tasks* to perform. A machine running a `bbot` instance in the agent mode is called a *build host*.

The actual building is performed in the virtual machines and/or containers that are executed on the build host. Inside virtual machines/containers, `bbot` is running in the *worker mode* and receives build tasks from its agent. Virtual machines and containers running a `bbot` instance in the worker mode are collectively called *build machines*.

Let's now examine the workflow in the other direction, that is, from a worker to a controller. Once a build machine is booted (by the agent), the worker inside connects to the TFTP server running on the build host and downloads the *build task manifest*. It then proceeds to perform the build task and uploads the *build result manifest* (which includes build logs) to the TFTP server.

Once an agent receives a build task for a specific build machine, it goes through the following steps. First, it creates a directory on its TFTP server with the *machine name* as its name and places the build task manifest inside. Next, it makes a throw-away snapshot of the build machine and boots it. After booting the build machine, the agent monitors the machine directory on its TFTP server for the build result manifest (uploaded by the worker once the build has completed). Once the result manifest is obtained, the agent shuts down the build machine and discards its snapshot.

To obtains a build task the agent polls via HTTP/HTTPS one or more controllers. Before each poll request the agent enumerates the available build machines and sends this information as part of the request. The controller responds with a build task manifest that identifies a specific build machine to use.

If the controller has higher-level controllers (for example, `brep`), then it aggregates the available build machines from its agents and polls these controllers (just as an agent would), forwarding build tasks to suitable agents. In this case we say that the *controller act as an agent*. The controller may also be configured to monitor build sources, such as SCM repositories, directly in which case it generates build tasks itself.

In this architecture the build results are propagated up the chain: from a worker, to its agent, to its controller, and so on. A controller that is the final destination of a build result uses email to notify interested parties of the outcome. For example, `brep` would send a notification to the package owner if the build failed. Similarly, a `bbot` controller that monitors a `git` repository would send an email to a committer if their commit caused a build failure. The email would include a link (normally HTTP/HTTPS) to the build logs hosted by the controller.

2.1 Configurations

The `bbot` architecture distinguishes between a *machine configuration* and a *build configuration*. The machine configuration captures the operating system, installed compiler toolchain, and so on. The same build machine may be used to "generate" multiple *build configurations*. For example, the same machine can normally be used to produce 32/64-bit and debug/release builds.

The machine configuration is *approximately* encoded in its *machine name*. The machine name is a list of components separated with `-`. Each component can contain alpha-numeric characters, underscores, dots, and pluses with the whole id being a portably-valid path component.

The encoding is approximate in a sense that it captures only what's important to distinguish in a particular `bbot` deployment.

The first component normally identifies the operating system and has the following recommended form:

```
[<arch>_][<class>_]<os>[_<version>]
```

For example:

```
windows
windows_10
windows_10.1607
i686_windows_xp
bsd_freebsd_10
linux_centos_6.2
linux_ubuntu_16.04
macos_10.12
```

The second component normally identifies the installed compiler toolchain and has the following recommended form:

```
<id>[<version>][<runtime>]
```

For example:

```
gcc
gcc_6
gcc_6.3
clang_3.9_libc++
clang_3.9_libstdc++
msvc_14
msvc_14u3
icc
```

Some examples of complete machine names:

```
windows_10-msvc_14u3
macos_10.12-clang
linux_ubuntu_16.04-gcc_6.3
```

Similarly, the build configuration is encoded in a *configuration name* using the same format. As described in Controller Logic, build configurations are generated from machine configurations. As a result, it usually makes sense to have the first component identify the operating systems and the second component – the toolchain with the rest identifying a particular build configuration. For example:

```
windows-vc_14-32-debug
linux-gcc_6-cross-arm-eabi
```

2.2 Machine Header Manifest

```
SYNOPSIS

id:      <machine-id>
name:    <machine-name>
summary: <string>
```

The build machine header manifest contains basic information about a build machine on the build host. A list of machine header manifests is sent by `bbot` agents to controllers.

`id: <machine-id>`
: The *machine-id* uniquely identifies a machine version/revision/build. For virtual machines this can be the disk image checksum. For a container this can be UUID that is re-generated every time a container filesystem is altered.

`name: <machine-name>`
: The machine name as described above.

`summary: <string>`
: A one-line description of the machine. For example:

```
id: windows_10-msvc_14-1.3
name: windows_10-msvc_14
summary: Windows 10 build 1607 with VC 14 update 3
```

2.3 Machine Manifest

```
SYNOPSIS

id:      <machine-id>
name:    <machine-name>
summary: <string>

type:    <machine-type>
mac:     <macaddr>
options: <machine-options>
```

The build machine manifest contains the complete description of a build machine on the build host (see the Build OS documentation for their origin and location). The machine manifest starts with the machine manifest header. All the header values must appear before any non-header values.

`type: <machine-type>`
The machine type. Valid values are `kvm` (QEMU/KVM virtual machine) and `nspawn` (`systemd-nspawn` container).

`mac: <macaddr>`
Optional fixed MAC address for the machine in the hexadecimal, comma-separated format. For example:

```
mac: de:ad:be:ef:de:ad
```

If it is not specified, then a random address is generated on the first machine bootstrap which is then reused for each build/re-bootstrap. Note that it you specify a fixed address, then the machine can only be used by a single `bbot` agent.

`options: <machine-options>`
Optional list of machine options. The exact semantics is machine type-dependent (see below). A single level of quotes (either single or double) is removed in each option before being passed on. Options can be separated with spaces or newlines.

For `kvm` machines, if this value is present, then it replaces the default network and disk configuration when starting the QEMU/KVM hypervisor The options are pre-processed by replacing the question mark in `ifname=?` and `mac=?` strings with the network interface and MAC address, respectively.

2.4 Task Manifest

```
SYNOPSIS

name:       <package-name>
version:    <package-version>
#location:  <package-url>
repository: <repository-url>
trust:      <repository-fp>

machine:       <machine-name>
target:        <target-triplet>
config:        <config-vars>
warning-regex: <warning-regexes>
```

The task manifest describes a build task. It consists of two groups of values. The first group defines the package to build. The second group defines the build configuration to use for building the package.

`name: <package-name>`
: Package name to test.

`version: <package-version>`
: Package version to test.

`repository: <repository-url>`
: The `bpkg` repository that contains the package and its dependencies.

`trust: <repository-fp>`
: The SHA256 repository certificate fingerprint to trust (see the `bpkg --trust` option for details). This value may be specified multiple times to establish the authenticity of multiple certificates. If the special `yes` value is specified, then all repositories will be trusted without authentication (see the `bpkg --trust-yes` option).

  Note that while the controller may return a task with `trust` values, whether they will be used is up to the agent's configuration. For example, some agents may only trust their internally-specified fingerprints to prevent the "man in the middle" type of attacks.

`machine: <machine-name>`
: The name of the build machine to use.

`target: <target-triplet>`
: The target triplet to build for.

  Compared to the autotools terminology, the `machine` value corresponds to `--build` (the machine we are building on) and `target` – to `--host` (the machine we are building for). While we use essentially the same *target triplet* format as autotools for `target`, it is not flexible enough for `machine`.

`config: <config-vars>`

Additional build system configuration variables.

A single level of quotes (either single or double) is removed in each variable before being passed to `bpkg`. For example, the following value:

```
config: config.cc.coptions="-O3 -stdlib='libc++'"
```

Will be passed to `bpkg` as the following (single) argument:

```
config.cc.coptions=-O3 -stdlib='libc++'
```

Variables can be separated with spaces or newlines.

`warning-regex: <warning-regexes>`

Additional regular expressions that should be used to detect warnings in the logs. Note that only the first 512 bytes of each log line is considered.

A single level of quotes (either single or double) is removed in each expression before being used for search. For example, the following value:

```
warning-regex: "warning C4\d{3}: "
```

Will be treated as the following (single) regular expression (with a trailing space):

```
warning C4\d{3}: 
```

Expressions can be separated with spaces or newlines. They will be added to the following default list of regular expressions that detect the `build2` toolchain warnings:

```
^warning:
^.+: warning:
```

Note that this built-in list also covers GCC and Clang warnings (for the English locale).

2.5 Result Manifest

```
SYNOPSIS

name:     <package-name>
version:  <package-version>

status:                 <status>
configure-status:       <status>
update-status:          <status>
test-status:            <status>
install-status:         <status>
test-installed-status:  <status>
uninstall-status:       <status>
```

```
configure-log:       <text>
update-log:          <text>
test-log:            <text>
install-log:         <text>
test-installed-log: <text>
uninstall-log:       <text>
```

The result manifest describes a build result.

`name: <package-name>`

Package name from the task manifest.

`version: <package-version>`

Package version from the task manifest.

`status: <status>`

An overall (cumulative) build result status. Valid values are:

```
success    # All operations completed successfully.
warning    # One or more operations completed with warnings.
error      # One or more operations completed with errors.
abort      # One or more operations were aborted.
abnormal   # One or more operations terminated abnormally.
```

The `abort` status indicates that the operation has been aborted by `bbot`, for example, because it was consuming too many resources and/or was taking too long. Note that a task can be aborted both by the `bbot` worker as well as the agent. In the later case the whole machine is shut down and no operation-specific status or logs will be included (@@ Maybe we should just include 'log:' with commands that start VM, for completeness?).

The `abnormal` status indicates that the operation has terminated abnormally, for example, due to the package manager or build system crash.

Note that the overall `status` value should appear before any per-operation `*-status` values.

`*-status: <status>`

A per-operation result status. Note that the `*-status` values should appear in the same order as the corresponding operations were performed and for each `*-status` there should be a corresponding `*-log`.

`*-log: <text>`

A per-operation result log. Note that the `*-log` values should appear last and in the same order as the corresponding `*-status` values.

2.6 Task Request Manifest

```
SYNOPSIS

agent:              <agent-name>
toolchain-name:     <name>
toolchain-version: <standard-version>
fingerprint:        <agent-fingerprint>
```

An agent (or controller acting as an agent) sends a task request to its controller via HTTP/HTTPS POST method (@@ URL/API endpoint). The task request starts with the task request manifest followed by a list of machine manifests.

`agent: <agent-name>`
: The name of the agent host (`hostname`). This should be unique in a particular `bbot` deployment.

`toolchain-name: <name>`
: The `build2` toolchain name being used by the agent.

`toolchain-version: <standard-version>`
: The `build2` toolchain version being used by the agent.

`fingerprint: <agent-fingerprint>`
: The SHA256 fingerprint of the agent's public key. An agent may be configured not to use the public key-based authentication in which case it does not include this value. However, the controller may be configured to require the authentication in which case it will respond with the 401 (unauthorized) HTTP status code.

2.7 Task Response Manifest

```
SYNOPSIS

session:    <session-id>
challenge:  <text>
result-url: <url>
```

A controller sends the task response manifest in response to the task request initiated by an agent. The response is delivered as a result of the POST method. The task response starts with the task response manifest optionally followed by a task manifest.

`session: <session-id>`
: An identifier assigned to this session by the controller. An empty value indicates that the controller has no tasks at this time in which case all the following values as well as the task manifest are absent.

`challenge: <string>`
: Random 64-character string (nonce) used to challenge the agent's private key. If present, then the agent must sign this string and include the signature in the result request.

The signature should be calculated by encrypting the string with the agent's private key and then base64-encoding the result.

`result-url: <url>`
The URL to post the result (upload) request to.

2.8 Result Request Manifest

```
SYNOPSIS

session:   <session-id>
challenge: <text>
```

On completion of a task an agent (or controller acting as an agent) sends a result (upload) request to the controller via HTTP/HTTPS POST method using the URL returned in the task response. The result request starts with the result request manifest followed by a result manifest. Note that there is no result response and only a successful but empty POST result is returned.

`session: <session-id>`
The session id as returned by the controller in the task response.

`challenge: <text>`
The answer to the private key challenge as posed by the controller in the task response. Must be present only if the challenge value was present in the task response.

2.9 Worker Logic

The `bbot` worker builds each package in a *build environment* that is established for a particular build target. The environment has three components: the execution environment (environment variables, etc), build system modules, and configuration variables.

Setting up of the environment is performed by an executable (script, batch file, etc). Specifically, upon receiving a build task, the worker obtains its target and looks for the environment setup executable with this name in a specific directory. If not found, then the worker looks for the executable called `default`. Not being able to locate the environment executable is an error.

Once the environment setup executable is determined, the worker re-executes itself as that executable passing to it as command line arguments the target name, the path to the `bbot` worker to be executed once the environment is setup, and any additional options that need to be propagated to the re-executed worker. The environment setup executable is executed in the build directory as its current working directory. The build directory contains the build task `manifest` file.

The environment setup executable sets up the necessary execution environment for example by adjusting `PATH` or running a suitable `vcvars` batch file. It then re-executes itself as the `bbot` worker passing to it as command line arguments (in addition to worker options) the list of build system modules (<env-modules>) and the list of configuration variables (<env-config-vars>). The environment setup executable must execute the bbot worker in the build directory as the current working directory.

The re-executed bbot worker then proceeds to test the package from the repository by executing the following commands (<>-values are from the task manifest and environment):

```
bpkg -v create <env-module> <config-vars> <env-config-vars>
bpkg -v add <repository-url>
bpkg -v fetch --trust <repository-fp>
bpkg -v build --yes --configure-only <package-name>/<package-version>
bpkg -v update <package-name>
bpkg -v test <package-name>
```

As an example, the following POSIX shell script can be used to setup the environment for building C and C++ packages with GCC 6 on most Linux distributions.

```
#!/bin/sh

# Environment setup script for C/C++ compilation with GCC 6.
#
# $1  - target
# $2  - bbot executable
# $3+ - bbot options

set -e # Exit on errors.

t="$1"
shift

if test -n "$t"; then
  echo "unknown target: $t" 1>&2
  exit 1
fi

exec "$@" cc config.c=gcc-6 config.cxx=g++-6
```

2.10 Controller Logic

A bbot controller that issues own build tasks maps available build machines (as reported by agents) to *build configurations* according to the buildtab configuration file. Blank lines and lines that start with # are ignored. All other lines in this file have the following format:

```
<machine-pattern> <config> <target> [<config-vars>] [<warning-regex>]
```

Where <machine-pattern> is filesystem wildcard pattern that is matched against available machine names, <config> is the configuration name, <target> is the build target, optional <config-vars> is a list of additional build system configuration variables, and optional <warning-regex> is a list of additional regular expressions that should be used to detect warnings in the logs.

Regular expressions must start with ~, to be distinguished from configuration variables. Note that `<config-vars>` and `<warning-regex>` lists have the same quoting semantics as in the `config` and the `warning-regex` values in the build task manifest. The matched machine name, the target, configuration variables, and regular expressions are included into the build task manifest.

Note that each machine name is matched against every pattern and all the patterns that match produce a configuration. If a machine does not match any pattern, then it is ignored (meaning that this controller is not interested in testing its packages with this machine). If multiple machines match the same pattern, then only a single configuration using any of the machines is produced (meaning that this controller considers these machines equivalent).

As an example, let's say we have a machine named `windows_10-vc_14u3`. If we wanted to test both 32 and 64-bit builds as well as debug and release, then we could have generated the following configurations:

```
windows*-vc_14* windows-vc_14-32-debug i686-microsoft-win32-msvc14.0 config.cc.coptions=/Z7 config.cc.loptions=/DEBUG ~"warning C4\d{3}: "

windows*-vc_14* windows-vc_14-32-release i686-microsoft-win32-msvc14.0 config.cc.coptions="/O2 /Oi" ~"warning C4\d{3}: "

windows*-vc_14* windows-vc_14-64-debug x86_64-microsoft-win32-msvc14.0 config.cc.coptions=/Z7 config.cc.loptions=/DEBUG ~"warning C4\d{3}: "

windows*-vc_14* windows-vc_14-64-release x86_64-microsoft-win32-msvc14.0 config.cc.coptions="/O2 /Oi" ~"warning C4\d{3}: "
```

As another example, let's say we have `linux_fedora_25-gcc_6` and `linux_ubuntu_16.04-gcc_6`. If all we cared about is testing GCC 6 64-bit builds on Linux, then our configurations could look like this:

```
linux*-gcc_6 linux-gcc_6-debug x86_64-linux-gnu config.cc.coptions=-g
linux*-gcc_6 linux-gcc_6-release x86_64-linux-gnu config.cc.coptions=-O3
```

If `<config-vars>` contains the `config.install.root` variable then, in addition to building and running tests, the `bbot` worker will also test installing and uninstalling each package. Furthermore, if the package contains the `tests` subdirectory that is a subproject, then the worker will additionally build and run tests against the installation.

Two types of installations can be tested: *system* and *private*. A system installation uses a well-known location, such as `/usr` or `/usr/local`, that will be searched by the compiler toolchain by default. A private installation uses a private directory, such as `/opt`, that will have to be explicitly mentioned to the compiler. While the system installation is usually preferable, it may not be always usable because of the potential conflicts with the already installed software, for example, by the system package manager.

As an example, the following two configurations could be used to test system and private installations:

```
linux*-gcc* linux-gcc-sys x86_64-linux-gnu config.install.root=/usr config.install.sudo=sudo
linux*-gcc* linux-gcc-prv x86_64-linux-gnu config.install.root=/tmp/install config.cc.poptions=-I/tmp/install/include config.cc.loptions=-L/tmp/install/lib config.bin.rpath=/tmp/install/lib
```

Note also that while building and run tests against the installation the worker makes the `bin` subdirectory of `config.install.root` the first entry in the `PATH` environment variable.