crcache Documentation

Release trunk

crcache contributors

October 27, 2013

CONTENTS

Contents:

CHAPTER

ONE

CRCACHE USERS GUIDE

1.1 Overview

crcache acts as a layer to obtain, use, and discard compute resources - be it virtual machines, chroots or even physical machines. This is a common requirement for testing environments, and having an abstraction layer allows a single project setup to scale in dramatically different ways just by the user reconfiguring their crcache config.

This manual covers both part planned works and implemented works. If something doesn't work please file a bug.

1.2 Requirements

- Python 2.6+ or 3.2+
- The 'extras' Python package.
- For testing a number of other packages (see setup.py).

1.3 Installation

Using pip is the easiest way to install crcache:

```
$ pip install crcache
```

1.4 Configuration

The default configuration is to have a single source `local` which runs commands locally.

1.4.1 Search path

Configuration is looked up in ~/.config/crcache and $(pwd)/.crcache. Where something is defined in both places, the first found definition wins, allowing local configuration to supercede any configuration supplied in a project (which might be version controlled and thus harder to change without side effects).

1.4.2 Sources

Each `source` is a subdirectory of a config root - `$root/sources/$sourcename`. Sources define how to provision one or more compute resources.

A source called `local` will replace the implicit definition of the local source.

The file `source.conf` is a .ini file that controls basic metadata for the source:

```
[DEFAULT]
; What sort of source is this?
type=[local|pool|ssh]
; Do not discard instances if less than this many are running.
; Defaults to 0 - avoids caching expensive resources w/out warning.
reserve=int
; Do not scale out beyond this many instances.
; Defaults to 0 - no limit.
maximum=int
; Override the concurrency of returned instances, rather than probing.
; Defaults to 0 - autoprobe.
concurrency=int
; For pools only
sources=sourcename,sourcename,...
; For ssh only
ssh_host=string
```

If a directory called `provision.d` exists as a sibling to `source.conf` then its contents will be run as they are provisioned (using run-parts). The resource name is supplied to the scripts as the first parameter - the script can call `crcache run` to execute commands on the resource.

Likewise for `discard.d` immediately before discarding an instance.

1.5 Command line

1.5.1 status

Provides details of sources and resources:

```
$ crcache status
source  cached  in-use max
local   0       1      1
pool    1       0      1

$ crcache status --query available pool
1
```

1.5.2 acquire

Checks a compute resource out for use:

```
$ crcache -s pool acquire
pool-0

$ crcache status pool
source  cached  in-use max
pool    0       1      1
```

1.5.3 run

Runs a command on a checked out resource:

```
$ crcache run pool-0 echo foo
foo
```

Get a shell on the resource:

```
$ crcache run pool-0
...
```

1.5.4 copy

Copies files into (or out of) the resource:

```
$ crcache cp /tmp/foo pool-0:/tmp
```

1.5.5 release

Returns a compute resource from use:

```
$ crcache release pool0
$ crcache status pool
source  cached  in-use max
pool    1       0      1
```

1.6 Internals

Each source stores the instances it has obtained and has cached in the crcache store, stored in $HOME/.cache/crcache/state.db.

1.7 API

The internal API is largely uninteresting for users - and see the DESIGN and DEVELOPER documentation if you are interested. That said, one possibly common need is creating additional source types, and so we cover that here.

Source types are looked up by looking for a python module with the same name in the `cr_cache.source.` package namespace. They can be installed as a third-party using namespace packages, or patched into the main crcache source tree. Source modules should include a `Source` class, which the source type loader looks for - you can subclass `source.AbstractSource` or just implement its contract. The loader will instantiate a `Source` instance with a `ConfigParser` and a `get_source` callback (which permits sources to layer on other sources).

Sources are responsible for four things:

- Making instances that can run commands.
- Assigning unique (to the crcache instance) ids for the instances.
- Discarding such instances.
- Running commands on the instances.

Other operations, such as enforcing a limit on the number of instances, caching of instances, are taken care of by crcache infrastructure.

CHAPTER

TWO

DESIGN / ARCHITECTURE OF CRCACHE

NB: This document describes intent as much as actuality. The code has precedence where things differ (though for to-implement features, the code may simply be not-yet-written).

2.1 Primary Goals

- Provide an abstraction layer so that test runners like testrepository or in general any process that needs to run in isolated or repeatable environments can do so without needing to re-invent the wheel.
- Make it possible for projects to have project specific config in-tree and machine/environment specific config maintained on the machine/environment.

2.2 Musts

- Be command line drivable (make it easy to use from many languages including the console).
- Be able to run things locally without any configuration.
- Let users do arbitrary operations to customise compute environment provisioning / resetting / reuse.
- Not require long lived daemon processes - when not being actively used, crcache should be gone. [Optional features may require a daemon].
- Be able to organise computing resources - not all things are equal. (No explict modelling needed - just provide a language for users to differentiate different resources).
- Be able to copy files in and out of the computing environment. While providing the basic run-a-command facility is enough to let sftp or rsync work, it is hard to implement safe temp file handling without a higher level interface.

2.3 Secondary Goals

Clean UI, predictable behaviour, small-tools feel.

2.4 Problem domain

Consider a generic parallelising test runner and a test suite that uses machine-scoped resources such as well known ports, database or message queue servers and fixed paths on disk.

Any attempt to parallelise that test suite will run into significant immediate problems - the code base will have to be made generic, so that test servers run on ephemeral ports, so that the test database uses random names (and possibly still require a mutex on schema operations in different databases... depending on the database engine) - all predictable resources need to be made unique. Failing to do that will cause sporadic failures in the test suite when the parallel execution happens to place contending tests opposite each other. The greater the parallelism, the worse the issue.

To run the test suite in parallel, it needs to be isolated. The most robust form of isolation is N separate machines with shared nothing, but thats a lot of overhead to manage. Virtual machines, containers or chroots offer varying degrees of less isolation but with correspondingly lower overhead for management. We can model any of these test environments - local processes, chroots, containers, VM's or even separate physical machines with one model.

For efficiency, it would be desirable to minimise repeated work involved with setting up and tearing down virtualised environments. It is from this aspect that the `cache` in the name `crcache` is drawn. The model needs to be compatible with sophisticated approaches such as lvm snapshots, golden cloud images and hot prepped instances.

2.5 Concepts

To deal with compute clouds (such as Openstack or EC2) we need to allow for configuration for a whole class of resources at once. This implies a minimum of two concepts:

1. A source of compute resources.
2. Individual resources.

Any given project will have its own configuration to perform on a machine (e.g. installing dependencies, checking out source code). This could imply a third concept - `Project`, but can also be represented as just a layered source of resources, where the layer consumes from the layer below and performs whatever configuration is needed.

After configuring a resource for a project, the resource is ready to be used. To mask latency or avoid repeated work, preparing multiple resources in advance may be useful, which also argues for a new concept - `resource pools`.

However, there isn't (yet) any clear important differentiator between a source of compute resources and a pool that draws from other sources - we can treat a pool as just another source. So, like project, pooled resources will exist but only as a specialised resource source.

It can be argued that sources like EC2 which require credentials and so on should be given two levels of configuration - global and per-project-binding. In the interest of minimising concepts, that is not done today.

Some resources can share local file trees very efficiently, e.g. via COW file systems, bind mounting, bind mounting with layered file sytems, or even cluster file systems. This offers huge performance benefits when used, so this becomes a necessary concept:

3. Filesystem exporting.

We need to let users run arbitrary code under crcaches control from time to time, so thats also a necessary concept:

4. Extension points.

The lifecycle of a resource, with all optimisations in place, will be something like:

1. Provision, either statically configured or dynamically via some API. [needs source, produces resource]
2. Perform per-project configuration and place into a pool ready for use. The pool might be a stopped lxc container, or a running but idle cloud instance. [needs resource, pool source, produces resource]

3. Take it out of the pool and perform per-revision configuration. [needs pool, produces allocated resource]
4. Run some commands on it / copy files to or from it. [needs allocated resource]
5. Reset it to pool-status. This might involve stopping it and doing an lvm rollback, unmounting an aufs filesystem from a chroot, or doing nothing. [needs allocated resource, discards resource]
6. Repeat 3-5 as needed.
7. Unprovision, either dynamically, or by a user removing the configuration data. [needs source, pooled resource]

2.6 Resource Source

2.6.1 Scale

Sources have a range of concurrency. Fixed resources have the lower and upper bounds the same, indicating that there is no way to discard such resources. However, they start out with none allocated. Sources with non-zero lower bounds could be preferentially used to fill pool requests.

2.6.2 Provision

Sources need an API call to obtain another resource from the source. Allowing users to run arbitrary code on the resource as it is obtained will allow significant flexability with little code overhead.

2.6.3 Discard

Sources need to be able to discard a resource they previously created. While perhaps a corner case, allowing users to run arbitrary code on the resource prior to discarding it is symmetrical and that helps predictability.

2.6.4 Local source

Runs commands locally. Possible configuration options:

- Explicit concurrency.
- Override CWD.
- Do a sudo call ?
- Make file copies not copy (e.g. cp -al, or symlink...)
- Can import filesystems by bind mounting or even just running in the right dir.

2.6.5 SSH source

Runs commands by sshing into a host. Possible configuration options:

- Host to ssh into
- Optional source to layer on? [permits bastion hosts] Raises the question of shared use of a bastion host - how to avoid locking other users out when the actual resource being used is behind the bastion host, while still not permitting the bastion host to be gc'd.
- Number of instances to export ?

2.6.6 Chroot source

Makes chroots. Configuration options:

- command line to instantiate a chroot
- command line to execute a command in a chroot
- control the user to run commands as
- import filesystems by bind mounting
- Layers on a base level source.
- Number of chroots to permit ?

2.6.7 LXC source

Make LXC containers. Same basic options as chroots.

2.6.8 Cloud source

- cloud provider credentials, machine image id.
- SSH private key to use to make connections.

2.6.9 Pool source

A pool backends onto other sources. Configuration:

- One or more sources
- Minimum scale - able to be dialed up higher than the sum of the minimum scale for the backend sources. (Dialing it lower would have no impact, because the backends would maintain their own minimums.

2.7 Compute Resource

2.7.1 Concurrency

Any given machine, be it virtual or physical, has an intrinsic degree of concurrency. This matters to users that are scheduling work - for instance, a test suite that has a natively parallel test runner might want to run one instance of it per machine, but be spread over several physical machines to get better concurrency. Something orchestrating runs with that runner would want to know N(machines) rather than N(cpus) when scheduling work. Conversely, a test runner that is itself serial and only ever uses one CPU per process might want to run some M processes per physical machine, where M is the number of actual cores in the machine.

We can expose the concurrency (ideally the effective cores, but as an approximation the number of cpu's the OS sees) to clients of crcache. If we choose not to expose this, users could just provision single-core resources everywhere, but that has its own inefficiencies and the more cores machines have the more getting this right will matter.

Users may want to control this - e.g. to deal with poor CPU topologies so offering an extension point to override (or perhaps mutate) the auto-detected value makes sense. OTOH users could just wrap crcache calls.

2.7.2 Running tasks

We need to be able to run tasks on a resource. To do that you need a network location, username and credentials. We can bundle those all up and offer a remote shell facility, with minimal loss of generality.

crcache is a choke point on command execution, so it can offer an extension point both before and after commands are run (and perhaps even wrap the input and output of commands). Uses for this are to fix up paths, environment variables, squelch noise at the source. However, most of the same capability can be done by wrapping crcache itself, so this should be a second-pass feature.

2.7.3 File handoffs

A common task will be synchronising some local file with the resource, and retrieving build products post-execution. While anything can be build on the run-a-task abstraction, offering direct file handling simplifies correctness for handling of temporary files, and makes debugging considerably easier for users. In particular, if there are extension points to influence task running, file transfer done on top of running tasks would be subject to the same side effects.

2.7.4 Filesytem imports

What sort of imports can this resource utilise?

- rsync
- bind mount
- others in future?

2.8 Code layout

One conceptual thing per module, packages for anything where multiple types are expected (e.g. cr_cache.commands, cr_cache.ui).

Generic driver code should not trigger lots of imports: code dependencies should be loaded when needed. For example, argument validation uses argument types that each command can import, so the core code doesn't need to know about all types.

The tests for the code in cr_cache.foo.bar is in cr_cache.tests.foo.test_bar. Interface tests for cr_cache.foo is in cr_cache.tests.foo.test___init__.

2.9 Key modules

2.9.1 cache

Responsible for arbitrating use of sources. Takes care to stay within limits, manage reserved resources etc.

2.9.2 source

Pluggable interface for supplying compute resources. Takes care of making, discarding, and running commands on compute resources.

2.9.3 ui

User interfaces.

2.9.4 commands

Tasks users can perform.

2.10 External integration

The command, ui, parsing etc objects should all be suitable for reuse from other programs - e.g. to provide a GUI or web status page with pool status.

CHAPTER

THREE

DEVELOPING CRCACHE

3.1 Releases

To do a release:

1. Update crcache/__init__.py to the new version.
2. Commit, make a signed tag.
3. Run *./setup.py sdist upload -s*.
4. Push the tag and trunk.

3.2 Hacking

(See also doc/DESIGN.rst).

The primary repository is https://github.com/rbtcollins/crcache. Please branch from there and use pull requests to submit changes. Bug tracking is the github bug tracker.

3.3 Coding style

Pep8. Be liberal with pylint. Pragmatism over purity.

Test everything that can be sensibly tested.

3.4 Tests

Can be run either with *./setup.py test* (which should install the needed dependencies) or *testr run* (if you have installed testrepository). If for some reason *setup.py test* does not install dependencies, they can be found by looking in `setup.py`.

3.5 Copyright

Contributions need to be dual licensed (see COPYING), but no copyright assignment or grants are needed.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*