Avocado Virt Test Compatibility Layer Documentation

Release 100.1

Lucas Meneghel Rodrigues

Sep 21, 2023

Contents

Contents:

CHAPTER 1

About Avocado-VT

Avocado-VT is a compatibility plugin that lets you execute virtualization related tests (then known as virt-test), with all conveniences provided by Avocado.

Its main purpose is to serve as an automated regression testing tool for virt developers, and for doing regular automated testing of virt technologies (provided you use it with the server testing infrastructure).

Avocado-VT aims to be a centralizing project for most of the virt functional and performance testing needs. We cover:

- Guest OS install, for both Windows (WinXP - Win7) and Linux (RHEL, Fedora, OpenSUSE and others through step engine mechanism)
- Serial output for Linux guests
- Migration, networking, timedrift and other types of tests

For the qemu subtests, we can do things like:

- Monitor control for both human and QMP protocols
- Build and use qemu using various methods (source tarball, git repo, rpm)
- Some level of performance testing can be made.
- The KVM unit tests can be run comfortably from inside virt-test, we do have full integration with the unittest execution

We support x86_64 hosts with hardware virtualization support (AMD and Intel), and Intel 32 and 64 bit guest operating systems.

1.1 About virt-test

Virt-test is the project that became Avocado-VT. It used to live under the Autotest umbrella, under:

http://github.com/autotest/virt-test

That repository is now frozen and only available at that location for historical purposes.

CHAPTER 2

Getting Started

The first step towards using Avocado-VT is, quite obviously, installing it.

2.1 Installing Avocado-VT

Avocado-VT is an Avocado plugin, therefore you are going to need both in order to be able to execute the tests.

Both are primarily written in Python, so a standard Python installation is possible and often preferable.

If you just want to use the plugin to run tests you might prefer to use packages from your system's package manager if available; this way non-python dependencies, esp. Avocado, are also taken care of automatically.

You can find more details about the Avocado installation here.

2.1.1 Installing via PIP

Pip is useful when it comes to python dependencies, but it fails in non-python ones. List of non-python requirements based on Fedora package names is:

```
$ dnf install xz tcpdump iproute iputils gcc glibc-headers nc git
```

Then you can get Avocado-VT via pip:

```
$ pip install git+https://github.com/avocado-framework/avocado-vt
```

Or by manually cloning it from github:

```
$ git clone https://github.com/avocado-framework/avocado-vt
$ cd avocado-vt
$ pip install .
```

It's recommended to use `pip` even for local install as it treats requirements differently and the use of `python setup.py install` might fail.

2.1.2 Installing via system package manager

Installing Avocado-VT on Fedora or Enterprise Linux is a matter of installing the *avocado-plugins-vt* package. Install it with:

```
$ yum install avocado-plugins-vt
```

Which takes care of all the dependencies (python and non-python ones).

2.1.3 Setup Avocado-VT with sources

If you intend use avocado from sources, clone it into the same parent directory as Avocado sources and use `make link` from the Avocado sources directory. Details about this can be found here.

2.2 Bootstrapping Avocado-VT

After the package, a bootstrap process must be run. Choose your test backend (qemu, libvirt, v2v, openvswitch, etc) and run the *vt-bootstrap* command. Example:

```
$ avocado vt-bootstrap --vt-type qemu
```

Note: If you don't intend to use `JeOS` and don't want to install the `xz` you can use `avocado vt-bootstrap --vt-type qemu --vt-guest-os $OS_OF_YOUR_CHOICE` which bypasses the `xz` check.

The output should be similar to:

```
12:02:10 INFO | qemu test config helper
12:02:10 INFO |
12:02:10 INFO | 1 - Updating all test providers
12:02:10 INFO |
12:02:10 INFO | 2 - Checking the mandatory programs and headers
12:02:10 INFO | /bin/xz OK
12:02:10 INFO | /sbin/tcpdump OK
...
12:02:11 INFO | /usr/include/asm/unistd.h OK
12:02:11 INFO |
12:02:11 INFO | 3 - Checking the recommended programs
12:02:11 INFO | /bin/qemu-kvm OK
12:02:11 INFO | /bin/qemu-img OK
12:02:11 INFO | /bin/qemu-io OK
...
12:02:33 INFO | 7 - Checking for modules kvm, kvm-intel
12:02:33 DEBUG| Module kvm loaded
12:02:33 DEBUG| Module kvm-intel loaded
12:02:33 INFO |
12:02:33 INFO | 8 - If you wish, you may take a look at the online docs for more info
12:02:33 INFO |
12:02:33 INFO | http://avocado-vt.readthedocs.org/
```

If there are missing requirements, please install them and re-run *vt-bootstrap*.

Note: Recommended programs might be needed for Avocado to correctly recognize test cases for your test backend or for test cases to run correctly.

Warning: When you bootstrap avocado-vt the parallel run of avocado nrunner will be disabled by default, because the avocado-vt doesn't support parallel tests. If you run test suite without vt tests, you can enable parallel run by *run.max_parallel_tasks* config variable.

2.3 First steps with Avocado-VT

Let's check if things went well by listing the Avocado plugins:

```
$ avocado plugins
```

That command should show the loaded plugins, and hopefully no errors. The relevant lines will be:

```
Plugins that add new commands (avocado.plugins.cli.cmd):
vt-bootstrap Avocado VT - implements the 'vt-bootstrap' subcommand
...
Plugins that add new options to commands (avocado.plugins.cli):
vt      Avocado VT/virt-test support to 'run' command
vt-list Avocado-VT/virt-test support for 'list' command
```

Then let's list the tests available with:

```
$ avocado list --vt-type qemu --verbose
```

This should list a large amount of tests (over 1900 virt related tests):

```
ACCESS_DENIED: 0
BROKEN_SYMLINK: 0
BUGGY: 0
INSTRUMENTED: 49
MISSING: 0
NOT_A_TEST: 27
SIMPLE: 3
VT: 1906
```

Note: If no test cases are listed make sure you installed recommended programs on your system, s. "Bootstrapping Avocado-VT".

Now let's run a virt test:

```
$ avocado run type_specific.io-github-autotest-qemu.migrate.default.tcp
JOB ID     : <id>
JOB LOG    : /home/<user>/avocado/job-results/job-2015-06-15T19.46-1c3da89/job.log
JOB HTML   : /home/<user>/avocado/job-results/job-2015-06-15T19.46-1c3da89/html/
↪results.html
TESTS      : 1
(1/1) type_specific.io-github-autotest-qemu.migrate.default.tcp: PASS (95.76 s)
```

(continues on next page)

(continued from previous page)

```
PASS       : 1
ERROR      : 0
FAIL       : 0
SKIP       : 0
WARN       : 0
INTERRUPT  : 0
TIME       : 95.76 s
```

If you have trouble executing the steps provided in this guide, you have a few options:

- Send an e-mail to the avocado mailing list.
- Open an issue on the avocado-vt github area.
- We also hang out on IRC (irc.oftc.net, #avocado).

CHAPTER 3

Writing Tests

This documentation aims to help you write virt tests of your own. It's organized to explain briefly the source structure, then writing simple tests, then doing more complex stuff, such as defining custom guests.

Contents:

3.1 Test Providers

Test providers are the conjunction of a loadable module mechanism inside Avocado-VT that can pull a directory that will provide tests, config files and any dependencies, and those directories. The design goals behind test providers are:

- Make it possible for other organizations to maintain test repositories, in other arbitrary git repositories.
- Stabilize API and enforce separation of core Avocado-VT functionality and tests.

The test provider spec is divided in Provider Layout and Definition files.

3.1.1 Test Provider Layout

```
.
|-- backend_1        -> Backend name. The actual name doesn't matter.
|   |-- cfg          -> Test config directory. Holds base files for the test runner.
|   |-- deps         -> Auxiliary files such as ELF files, Windows executables,␣
↪images that tests need.
|   |-- provider_lib -> Shared libraries among tests.
|   `-- tests        -> Python test files.
|       `-- cfg      -> Config files for tests.
`-- backend_2
    |-- cfg
    |-- deps
    |-- provider_lib
    `-- tests
        `-- cfg
```

In fact, Avocado-VT libraries are smart enough to support arbitrary organization of python and config files inside the 'tests' directory. You don't need to name the top level sub directories after backend names, although that certainly makes things easier. The term 'backend' is used to refer to the supported virtualization technologies by Avocado-VT. As of this writing, the backends known by Avocado-VT are:

- generic (tests that run in multiple backends)
- qemu
- openvswitch
- libvirt
- v2v
- libguestfs
- lvsb

The reason why you don't need to name the directories after the backend names is that you can configure a test definition file to point out any dir name. We'll get into

3.1.2 Types of Test Providers

Each test provider can be either a local filesystem directory, or a subdirectory of a git repository. Of course, the git repo subdirectory can be the repo root directory, but one of the points of the proposal is that people can hold Avocado-VT providers inside git repos of other projects. Say qemu wants to maintain its own provider, they can do this by holding the tests, say, inside a tests/avocado_vt subdirectory inside qemu.git.

3.1.3 Test Provider definition file

The main Avocado-VT suite needs a way to know about test providers. It does that by scanning definition files inside the 'test-providers.d' sub directory. Definition files are *config parser files <http://docs.python.org/2/library/configparser.html>* that encode information from a test provider. Here's an example structure of a test provider file:

```
[provider]

# Test provider URI (default is a git repository, fallback to standard dir)
uri: git://git-provider.com/repo.git
#uri: /path-to-my-git-dir/repo.git
#uri: http://bla.com/repo.git
#uri: file://usr/share/tests

# Optional git branch (for git repo type)
branch: master

# Optionall git commit reference (tag or sha1)
ref: e44231e88300131621586d24c07baa8e627de989

# Pubkey: File containing public key for signed tags (git)
pubkey: example.pub

# What follows is a sequence of sections for any backends that this test
# provider implements tests for. You must specify the sub directories of
# each backend dir, reason why the subdir names can be arbitrary.
```

(continues on next page)

(continued from previous page)

```
[qemu]
# Optional subdir (place inside repo where the actual tests are)
# This is useful for projects to keep virt tests inside their
# (larger) test repos. Defaults to ''.
subdir: src/tests/qemu/

[agnostic]
# For each test backend, you may have different sub directories
subdir: src/tests/generic/
```

Example of a default Avocado-VT provider file:

```
[provider]
uri: https://github.com/autotest/tp-qemu.git
[generic]
subdir: generic/
[qemu]
subdir: qemu/
[openvswitch]
subdir: openvswitch/
```

Let's say you want to use a directory in your file system (/usr/share/tests/virt-test):

```
[provider]
uri: file://usr/share/tests/
[generic]
subdir: virt-test/generic/
[qemu]
subdir: virt-test/qemu/
[openvswitch]
subdir: virt-test/openvswitch/
```

3.2 Development workflow after the Repository Split

1. Fork the test provider you want to contribute to in github

https://help.github.com/articles/fork-a-repo

2. Clone the forked repository. In this example, we'll assume you cloned the forked repo to

```
/home/user/code/tp-libvirt
```

3. Add a file in `~/avocado/data/avocado-vt/test-providers.d`, with a name you like. We'll assume you chose

```
user-libvirt.ini
```

4. Contents of user-libvirt.ini:

```
[provider]
uri: file:///home/user/code/tp-libvirt
[libvirt]
subdir: libvirt/
[libguestfs]
```

(continues on next page)

(continued from previous page)

```
subdir: libguestfs/
[lvsb]
subdir: lvsb/
[v2v]
subdir: v2v/
```

5. This should be enough. Now, when you use `--list-tests`, you'll be able to see entries like:

```
...
1 user-libvirt.unattended_install.cdrom.extra_cdrom_ks.default_install.aio_native
2 user-libvirt.unattended_install.cdrom.extra_cdrom_ks.default_install.aio_threads
3 user-libvirt.unattended_install.cdrom.extra_cdrom_ks.perf.aio_native
...
```

6. Modify tests, or add new ones to your heart's content. When you're happy with your changes, you may create branches and send us pull requests.

3.3 Writing your own avocado VT test

In this article, we'll talk about:

1. Where the test files are located
2. Write a simple test file
3. Try out your new test, send it to the mailing list

3.3.1 Write our own 'uptime' test - Step by Step procedure

Now, let's go and write our uptime test, which only purpose in life is to pick up a living guest, connect to it via ssh, and return its uptime.

1. First we need to locate our provider directory. It's inside Avocado *data* directory (*avocado config –datadir*), usually in *~/avocado/data/avocado-vt*. We are going to write a generic *tp-qemu* test, so let's move into the right git location:

```
$ cd $AVOCADO_DATA/avocado-vt/test-providers.d/downloads/io-github-autotest-qemu
```

2. Our uptime test won't need any qemu specific feature. Thinking about it, we only need a vm object and establish an ssh session to it, so we can run the command. So we can store our brand new test under `generic/tests`:

```
$ touch generic/tests/uptime.py
$ git add generic/tests/uptime.py
```

3. OK, so that's a start. So, we have *at least* to implement a function `run`. Let's start with it and just put the keyword pass, which is a no op. Our test will be like:

```
def run(test, params, env):
    """
    Docstring describing uptime.
    """
    pass
```

4. Now, what is the API we need to grab a VM from our test environment? Our env object has a method, `get_vm`, that will pick up a given vm name stored in our environment. Some of them have aliases. `main_vm` contains the name of the main vm present in the environment, which is, most of the time, `vm1`. `env.get_vm` returns a vm object, which we'll store on the variable vm. It'll be like this:

```
def run(test, params, env):
    """
    Docstring describing uptime.
    """
    vm = env.get_vm(params["main_vm"])
```

5. A vm object has lots of interesting methods, which we plan on documenting them more thoroughly, but for now, we want to ensure that this VM is alive and functional, at least from a qemu process standpoint. So, we'll call the method `verify_alive()`, which will verify whether the qemu process is functional and if the monitors, if any exist, are functional. If any of these conditions are not satisfied due to any problem, an exception will be thrown and the test will fail. This requirement is because sometimes due to a bug the vm process might be dead on the water, or the monitors are not responding:

```
def run(test, params, env):
    """
    Docstring describing uptime.
    """
    vm = env.get_vm(params["main_vm"])
    vm.verify_alive()
```

6. Next step, we want to log into the vm. The vm method that does return a remote session object is called `wait_for_login()`, and as one of the parameters, it allows you to adjust the timeout, that is, the time we want to wait to see if we can grab an ssh prompt. We have top level variable `login_timeout`, and it is a good practice to retrieve it and pass its value to `wait_for_login()`, so if for some reason we're running on a slower host, the increase in one variable will affect all tests. Note that it is completely OK to just override this value, or pass nothing to `wait_for_login()`, since this method does have a default timeout value. Back to business, picking up login timeout from our dict of parameters:

```
def run(test, params, env):
    """
    Docstring describing uptime.
    """
    vm = env.get_vm(params["main_vm"])
    vm.verify_alive()
    timeout = float(params.get("login_timeout", 240))
```

7. Now we'll call `wait_for_login()` and pass the timeout to it, storing the resulting session object on a variable named session:

```
def run(test, params, env):
    """
    Docstring describing uptime.
    """
    vm = env.get_vm(params["main_vm"])
    vm.verify_alive()
    timeout = float(params.get("login_timeout", 240))
    session = vm.wait_for_login(timeout=timeout)
```

8. Avocado-VT will do its best to grab this session, if it can't due to a timeout or other reason it'll throw a failure, failing the test. Assuming that things went well, now you have a session object, that allows you to type in commands on your guest and retrieve the outputs. So most of the time, we can get the output of these commands through the method `cmd()`. It will type in the command, grab the stdin and stdout, return them so you can store

it in a variable, and if the exit code of the command is != 0, it'll throw a aexpect.ShellError?. So getting the output of the unix command uptime is as simple as calling `cmd()` with 'uptime' as a parameter and storing the result in a variable called uptime:

```
def run(test, params, env):
    """
    Docstring describing uptime.
    """
    vm = env.get_vm(params["main_vm"])
    vm.verify_alive()
    timeout = float(params.get("login_timeout", 240))
    session = vm.wait_for_login(timeout=timeout)
    uptime = session.cmd('uptime')
```

Warning: Some guests OS's do not respect terminal `echo` setting, corrupting the output. There are some workaround described in github issue#231.

9. If you want to just print this value so it can be seen on the test logs, just log the value of uptime using the logging library. Since that is all we want to do, we may close the remote connection, to avoid ssh/rss sessions lying around your test machine, with the method `close()`.

```
def run(test, params, env):
    """
    Docstring describing uptime.
    """
    vm = env.get_vm(params["main_vm"])
    vm.verify_alive()
    timeout = float(params.get("login_timeout", 240))
    session = vm.wait_for_login(timeout=timeout)
    uptime = session.cmd('uptime')
    logging.info("Guest uptime result is: %s", uptime)
    session.close()
```

10. Note that all failures that might happen here are implicitly handled by the methods called. If a test went from its beginning to its end without unhandled exceptions, avocado assumes the test automatically as **PASS**, *no need to mark a test as explicitly passed*. If you have explicit points of failure, for more complex tests, you might want to mark it explicitly. `test.cancel` makes the test **CANCEL** (it is a new feature which not supported in avocado 36lts, you may want to use `test.skip` to make the test **SKIP** to achieve the similar purpose), `test.error` makes the test **Error**, and `test.fail` makes the test **Fail**. And, in recent Avocado version (since commit 7ecf09fa), three exceptions: `TestFail`, `TestError`, and `TestCancel` were added to avocado namespace, so you can import and use them appropriately. Note, people should not import `exceptions` from `avocado.core` to raise them in test case, see avocado doc for more details. *BTW, check the uptime makes no sense, but let's continue this example for test status explanation*:

```
def run(test, params, env):
    """
    Docstring describing uptime.
    """
    vm = env.get_vm(params["main_vm"])
    vm.verify_alive()
    timeout = float(params.get("login_timeout", 240))
    session = vm.wait_for_login(timeout=timeout)
    uptime = session.cmd('uptime')
    logging.info("Guest uptime result is: %s", uptime)
    session.close()
```

(continues on next page)

(continued from previous page)

```
    expected_cancel_msg = '0 min'
    expected_err_msg = '1 day'
    expected_fail_msg = '10 days'
    if expected_cancel_msg in uptime:
        test.cancel('Cancel message')
    if expected_err_msg in uptime:
        test.error('Error message')
    if expected_fail_msg in uptime:
        test.fail('Fail message')
```

11. Now, I deliberately introduced a bug on this code just to show you guys how to use some tools to find and remove trivial bugs on your code. I strongly encourage you guys to check your code with the *inspektor* tool. This tool uses pylint to catch bugs on test code. You can install inspektor by adding the COPR repo https://copr.fedoraproject.org/coprs/lmr/Autotest/ and doing

```
$ yum install inspektor
```

After you're done, you can run it:

```
$ inspekt lint generic/tests/uptime.py
************* Module generic.tests.uptime
E0602: 10,4: run: Undefined variable 'logging'
Pylint check fail: generic/tests/uptime.py
Syntax check FAIL
```

12. Ouch. So there's this undefined variable called logging on line 10 of the code. It's because I forgot to import the logging library, which is a python library to handle info, debug, warning messages. Let's Fix it and the code becomes:

```
import logging

def run(test, params, env):
    """
    Docstring describing uptime.
    """
    vm = env.get_vm(params["main_vm"])
    vm.verify_alive()
    timeout = float(params.get("login_timeout", 240))
    session = vm.wait_for_login(timeout=timeout)
    uptime = session.cmd("uptime")
    logging.info("Guest uptime result is: %s", uptime)
    session.close()
```

13. Let's re-run `inspektor` to see if it's happy with the code generated:

```
$ inspekt lint generic/tests/uptime.py
Syntax check PASS
```

14. So we're good. Nice! Now, as good indentation does matter to python, *inspekt indent* will fix indentation problems, and cut trailing whitespaces on your code. Very nice for tidying up your test before submission:

```
$ inspekt indent generic/tests/uptime.py
```

15. Now, you can test your code. When listing the qemu tests your new test should appear in the list (or shouldn't it?):

```
$ avocado list uptime
```

16. There is one more thing to do. Avocado-vt does not walk the directories, it uses *Cartesian config* to define test and all possible variants of tests. To add our test to *Cartesian config* we need yet another file:

```
$ touch generic/tests/cfg/uptime.cfg
$ git add generic/tests/cfg/uptime.cfg
```

17. The file might look like this:

```
- uptime:
    virt_test_type = qemu libvirt
    type = uptime
```

    where the *virt_test_type* specifies what backends can run this test and *type* specifies the test file. The *.py* will be appended and it'll be searched for in the usual location.

18. For the second time, let's try to discover the test:

```
$ avocado list uptime
```

19. OK still not there. We need to propagate the change to the actual config by running *vt-bootstrap*:

```
$ avocado vt-bootstrap
```

20. And now you'll finally see the test:

```
$ avocado list uptime
```

21. Now, you can run your test to see if everything went well:

```
$ avocado run --vt-type qemu uptime
```

22. OK, so now, we have something that can be git committed and sent to the mailing list (partial):

```
diff --git a/generic/tests/uptime.py b/generic/tests/uptime.py
index e69de29..65d46fa 100644
--- a/tests/uptime.py
+++ b/tests/uptime.py
@@ -0,0 +1,13 @@
+import logging
+
+def run(test, params, env):
+    """
+    Docstring describing uptime.
+    """
+    vm = env.get_vm(params["main_vm"])
+    vm.verify_alive()
+    timeout = float(params.get("login_timeout", 240))
+    session = vm.wait_for_login(timeout=timeout)
+    uptime = session.cmd("uptime")
+    logging.info("Guest uptime result is: %s", uptime)
+    session.close()
```

23. Oh, we forgot to add a decent docstring description. So doing it:

```
import logging

def run(test, params, env):

    """
    Uptime test for virt guests:

    1) Boot up a VM.
    2) Establish a remote connection to it.
    3) Run the 'uptime' command and log its results.

    :param test: QEMU test object.
    :param params: Dictionary with the test parameters.
    :param env: Dictionary with test environment.
    """

    vm = env.get_vm(params["main_vm"])
    vm.verify_alive()
    timeout = float(params.get("login_timeout", 240))
    session = vm.wait_for_login(timeout=timeout)
    uptime = session.cmd("uptime")
    logging.info("Guest uptime result is: %s", uptime)
    session.close()
```

24. git commit signing it, put a proper description, then send it with git send-email. Profit!

3.4 Defining New Guests

Let's say you have a guest image that you've carefully prepared, and the JeOS just doesn't cut it. Here's how you add new guests:

3.4.1 Linux Based Custom Guest

If your guest is Linux based, you can add a config file snippet describing your test (We have a bunch of pre-set values for linux in the default config).

The drop in directory is

```
shared/cfg/guest-os/Linux/LinuxCustom
```

You can add, say, foo.cfg to that dir with the content:

```
- FooLinux:
    image_name = images/foo-linux
```

Which would make it possible to specify this custom guest using

```
$ avocado run migrate..tcp --vt-type qemu --vt-guest-os LinuxCustom.FooLinux
JOB ID     : 44a399b427c51530ba2fcc37087c100917e1dd8a
JOB LOG    : /home/lmr/avocado/job-results/job-2015-07-29T03.47-44a399b/job.log
JOB HTML   : /home/lmr/avocado/job-results/job-2015-07-29T03.47-44a399b/html/results.
↪html
TESTS      : 3
(1/3) type_specific.io-github-autotest-qemu.migrate.default.tcp: PASS (31.34 s)
```

(continues on next page)

(continued from previous page)

```
(2/3) type_specific.io-github-autotest-qemu.migrate.with_set_speed.tcp: PASS (26.99 s)
(3/3) type_specific.io-github-autotest-qemu.migrate.with_reboot.tcp: PASS (46.40 s)
RESULTS    : PASS 3 | ERROR 0 | FAIL 0 | SKIP 0 | WARN 0 | INTERRUPT 0
TIME       : 104.73 s
```

Provided that you have a file called images/foo-linux.qcow2, if using the qcow2 format image.

Other useful params to set (not an exhaustive list):

```
# shell_prompt is a regexp used to match the prompt on aexpect.
# if your custom os is based of some distro listed in the guest-os
# dir, you can look on the files and just copy shell_prompt
shell_prompt = [*]$
# If you plan to use a raw device, set image_device = yes
image_raw_device = yes
# Password of your image
password = 123456
# Shell client used (may be telnet or ssh)
shell_client = ssh
# Port were the shell client is running
shell_port = 22
# File transfer client
file_transfer_client = scp
# File transfer port
file_transfer_port = 22
```

3.4.2 Windows Based Custom Guest

If your guest is Linux based, you can add a config file snippet describing your test (We have a bunch of pre-set values for linux in the default config).

The drop in directory is

```
shared/cfg/guest-os/Windows/WindowsCustom
```

You can add, say, foo.cfg to that dir with the content:

```
- FooWindows:
    image_name = images/foo-windows
```

Which would make it possible to specify this custom guest using

```
$ avocado run migrate..tcp --vt-type qemu --vt-guest-os WindowsCustom.FooWindows
```

Provided that you have a file called images/foo-windows.qcow2.

Other useful params to set (not an exaustive list):

```
# If you plan to use a raw device, set image_device = yes
image_raw_device = yes
# Attention: Changing the password in this file is not supported,
# since files in winutils.iso use it.
username = Administrator
password = 1q2w3eP
```

CHAPTER 4

Install Optional Packages

Some packages are not set in the Avocado-VT as hard dependencies, because they may only be required depending on specific use cases.

If you run into problems while running specific tests, please verify if installing the mentioned packages fixes your problem.

4.1 Fedora and EL

Install the following packages:

1. Install a toolchain in your host, which you can do with Fedora and RHEL with:

```
$ yum groupinstall "Development Tools"
```

1. Install tcpdump, necessary to determine guest IPs automatically

```
$ yum install tcpdump
```

1. Install nc, necessary to get output from the serial device and other qemu devices

```
$ yum install nmap-ncat
```

1. Install the xz file archiver so you can uncompress the JeOS [2] image.

```
$ yum install xz
```

1. Install the autotest-framework package, to provide the needed autotest libs.

```
$ yum install --enablerepo=updates-testing autotest-framework
```

#. Install the fakeroot package, if you want to install from the CD Ubuntu and Debian servers without requiring root:

```
$ yum install fakeroot
```

If you don't install the autotest-framework package (say, your distro still doesn't have autotest packages, or you don't want to install the rpm), you'll have to clone an autotest tree and export this path as the AUTOTEST_PATH variable, both as root and as your regular user. One could put the following on their ~/.bashrc file:

```
$ export AUTOTEST_PATH="/path/to/autotest"
```

where this AUTOTEST_PATH will guide the run script to set up the needed libraries for all tests to work.

For other packages:

```
$ yum install git
```

So you can checkout the source code. If you want to test the distro provided qemu-kvm binary, you can install:

```
$ yum install qemu-kvm qemu-kvm-tools
```

To run libvirt tests, it's required to install the virt-install utility, for the basic purpose of building and cloning virtual machines.

```
$ yum install virt-install
```

It's useful to also install:

```
$ yum install python-imaging
```

Not vital, but very handy to do imaging conversion from ppm to jpeg and png (allows for smaller images).

4.1.1 Tests that are not part of the default JeOS set

If you want to run guest install tests, you need to be able to create floppies and isos to hold kickstart files:

```
$ yum install mkisofs
```

For newer distros, such as Fedora, you'll need:

```
$ yum install genisoimage
```

Both packages provide the same functionality, needed to create iso images that will be used during the guest installation process. You can also execute

4.1.2 Network tests

Last but not least, now we depend on libvirt to provide us a stable, working bridge. * By default, the kvm test uses user networking, so this is not entirely necessary. However, non root and user space networking make a good deal of the hardcode networking tests to not work. If you might want to use bridges eventually:

```
$ yum install libvirt bridge-utils
```

Make sure libvirtd is started:

```
$ service libvirtd start
```

Make sure the libvirt bridge shows up on the output of brctl show:

```
$ brctl show
bridge name bridge id       STP enabled interfaces
virbr0      8000.525400678eec   yes     virbr0-nic
```

4.2 Debian

Keep in mind that the current autotest package is a work in progress. For the purposes of running virt-tests it is fine, but it needs a lot of improvements until it can become a more 'official' package.

The autotest debian package repo can be found at https://launchpad.net/~lmr/+archive/autotest, and you can add the repos on your system putting the following on /etc/apt/sources.list:

```
$ deb http://ppa.launchpad.net/lmr/autotest/ubuntu raring main
$ deb-src http://ppa.launchpad.net/lmr/autotest/ubuntu raring main
```

Then update your software list:

```
$ apt-get update
```

This has been tested with Ubuntu 12.04, 12.10 and 13.04.

Install the following packages:

1. Install the autotest-framework package, to provide the needed autotest libs.

```
$ apt-get install autotest
```

1. Install the xz-utils file archiver so you can uncompress the JeOS [2] image.

```
$ apt-get install xz-utils
```

1. Install tcpdump, necessary to determine guest IPs automatically

```
$ apt-get install tcpdump
```

1. Install nc, necessary to get output from the serial device and other qemu devices

```
$ apt-get install netcat-openbsd
```

1. Install a toolchain in your host, which you can do on Debian and Ubuntu with:

```
$ apt-get install build-essential
```

#. Install fakeroot if you want to install from CD debian and ubuntu, not requiring root:

```
$ apt-get install fakeroot
```

So you install the core autotest libraries to run the tests.

If you don't install the autotest-framework package (say, your distro still doesn't have autotest packages, or you don't want to install the rpm), you'll have to clone an autotest tree and export this path as the AUTOTEST_PATH variable, both as root and as your regular user. One could put the following on their ~/.bashrc file:

```
$ export AUTOTEST_PATH="/path/to/autotest"
```

where this AUTOTEST_PATH will guide the run script to set up the needed libraries for all tests to work.

For other packages:

```
$ apt-get install git
```

So you can checkout the source code. If you want to test the distro provided qemu-kvm binary, you can install:

```
$ apt-get install qemu-kvm qemu-utils
```

To run libvirt tests, it's required to install the virt-install utility, for the basic purpose of building and cloning virtual machines.

```
$ apt-get install virtinst
```

To run all tests that involve filedescriptor passing, you need python-all-dev. The reason is, this test suite is compatible with python 2.4, whereas a std lib to pass filedescriptors was only introduced in python 3.2. Therefore, we had to introduce a C python extension that is compiled on demand.

```
$ apt-get install python-all-dev.
```

It's useful to also install:

```
$ apt-get install python-imaging
```

Not vital, but very handy to do imaging conversion from ppm to jpeg and png (allows for smaller images).

4.2.1 Tests that are not part of the default JeOS set

If you want to run guest install tests, you need to be able to create floppies and isos to hold kickstart files:

```
$ apt-get install genisoimage
```

4.2.2 Network tests

Last but not least, now we depend on libvirt to provide us a stable, working bridge. * By default, the kvm test uses user networking, so this is not entirely necessary. However, non root and user space networking make a good deal of the hardcode networking tests to not work. If you might want to use bridges eventually:

```
$ apt-get install libvirt-bin python-libvirt bridge-utils
```

Make sure libvirtd is started:

```
$ service libvirtd start
```

Make sure the libvirt bridge shows up on the output of brctl show:

```
$ brctl show
bridge name bridge id       STP enabled interfaces
virbr0      8000.525400678eec   yes     virbr0-nic
```

CHAPTER 5

Listing guests

If you want to see all guests defined, you can use

```
$ avocado list --vt-type [test type] --vt-list-guests
```

This will generate a list of possible guests that can be used for tests, provided that you have an image with them. The list will show which guests don't have an image currently available. If you did perform the usual bootstrap procedure, only JeOS.17.64 will be available.

Now, let's assume you have the image for another guest. Let's say you've installed Fedora 17, 64 bits, and that –list-guests shows it as downloaded

```
$ avocado list --vt-type qemu --vt-list-guests
...
Linux.CentOS.6.6.i386.i440fx (missing centos66-32.qcow2)
Linux.CentOS.6.6.x86_64.i440fx (missing centos66-64.qcow2)
```

You can list all the available tests for Fedora.17.64 (you must use the exact string printed by the test, minus obviously the index number, that's there only for informational purposes:

```
$ avocado list --vt-type qemu --vt-guest-os Linux.CentOS.6.6.i386.i440fx --verbose
...
VT           io-github-autotest-qemu.trans_hugepage.base
VT           io-github-autotest-qemu.trans_hugepage.defrag
VT           io-github-autotest-qemu.trans_hugepage.swapping
VT           io-github-autotest-qemu.trans_hugepage.relocated
VT           io-github-autotest-qemu.trans_hugepage.migration
VT           io-github-autotest-qemu.trans_hugepage.memory_stress
VT           io-github-autotest-qemu.ntpd
VT           io-github-autotest-qemu.clock_getres
VT           io-github-autotest-qemu.autotest_regression
VT           io-github-autotest-qemu.shutdown

ACCESS_DENIED: 0
BROKEN_SYMLINK: 0
```

(continues on next page)

(continued from previous page)

```
BUGGY: 0
FILTERED: 0
INSTRUMENTED: 52
MISSING: 0
NOT_A_TEST: 27
SIMPLE: 3
VT: 2375
```

Then you can execute one in particular. It's the same idea, just copy the individual test you want and run it:

```
$ avocado run balloon_check --vt-type qemu --vt-guest-os Fedora.21
```

And it'll run that particular test.

CHAPTER 6

Advanced Topics and Maintenance

Contents:

6.1 How tests are run

When running tests Avocado-VT will:

1) Get a dict with test parameters
2) Based on these params, prepare the environment - create or destroy vm instances, create/check disk images, among others
3) Execute the test itself, that will use several of the params defined to carry on with its operations, that usually involve: - If a test did not raise an exception, it PASSed - If a test raised a TestFail exception, it FAILed. - If a test raised a TestNAError, it SKIPPED. - Otherwise, it ERRORed.
4) Based on what happened during the test, perform cleanup actions, such as killing vms, and remove unused disk images.

The list of parameters is obtained by parsing a set of configuration files The command line options usually modify even further the parser file, so we can introduce new data in the config set.

6.2 Cartesian Configuration

Cartesian Configuration is a highly specialized way of providing lists of key/value pairs within combination's of various categories. The format simplifies and condenses highly complex multidimensional arrays of test parameters into a flat list. The combinatorial result can be filtered and adjusted prior to testing, with filters, dependencies, and key/value substitutions.

The parser relies on indentation, and is very sensitive to misplacement of tab and space characters. It's highly recommended to edit/view Cartesian configuration files in an editor capable of collapsing tab characters into four space characters. Improper attention to column spacing can drastically affect output.

6.2.1 Keys and values

Keys and values are the most basic useful facility provided by the format. A statement in the form <key> = <value> sets <key> to <value>. Values are strings, terminated by a linefeed, with surrounding quotes completely optional (but honored). A reference of descriptions for most keys is included in section Configuration Parameter Reference. The key will become part of all lower-level (i.e. further indented) variant stanzas (see section *variants*). However, key precedence is evaluated in top-down or 'last defined' order. In other words, the last parsed key has precedence over earlier definitions.

6.2.2 Variants

A 'variants' stanza is opened by a 'variants:' statement. The contents of the stanza must be indented further left than the 'variants:' statement. Each variant stanza or block defines a single dimension of the output array. When a Cartesian configuration file contains two variants stanzas, the output will be all possible combination's of both variant contents. Variants may be nested within other variants, effectively nesting arbitrarily complex arrays within the cells of outside arrays. For example:

```
variants:
    - one:
        key1 = Hello
    - two:
        key2 = World
    - three:
variants:
    - four:
        key3 = foo
    - five:
        key3 = bar
    - six:
        key1 = foo
        key2 = bar
```

While combining, the parser forms names for each outcome based on prepending each variant onto a list. In other words, the first variant name parsed will appear as the left most name component. These names can become quite long, and since they contain keys to distinguishing between results, a 'short-name' key is also used. For example, running `cartesian_config.py` against the content above produces the following combinations and names:

```
dict    1:  four.one
dict    2:  four.two
dict    3:  four.three
dict    4:  five.one
dict    5:  five.two
dict    6:  five.three
dict    7:  six.one
dict    8:  six.two
dict    9:  six.three
```

Variant shortnames represent the <TESTNAME> value used when results are recorded (see section Job Names and Tags. For convenience variants who's name begins with a '@' do not prepend their name to 'short-name', only 'name'. This allows creating 'shortcuts' for specifying multiple sets or changes to key/value pairs without changing the results directory name. For example, this is often convenient for providing a collection of related pre-configured tests based on a combination of others.

6.2.3 Named variants

Named variants allow assigning a parseable name to a variant set. This enables an entire variant set to be used for in *filters*. All output combinations will inherit the named variant key, along with the specific variant name. For example:

```
variants var1_name:
     - one:
         key1 = Hello
     - two:
         key2 = World
     - three:
variants var2_name:
     - one:
         key3 = Hello2
     - two:
         key4 = World2
     - three:

only (var2_name=one).(var1_name=two)
```

Results in the following outcome when parsed with `cartesian_config.py -c`:

```
dict    1:  (var2_name=one).(var1_name=two)
      dep = []
      key2 = World         # variable key2 from variants var1_name and variant two.
      key3 = Hello2        # variable key3 from variants var2_name and variant one.
      name = (var2_name=one).(var1_name=two)
      shortname = (var2_name=one).(var1_name=two)
      var1_name = two      # variant name in same namespace as variables.
      var2_name = one      # variant name in same namespace as variables.
```

Named variants could also be used as normal variables.:

```
variants guest_os:
     - fedora:
     - ubuntu:
variants disk_interface:
     - virtio:
     - hda:
```

Which then results in the following:

```
dict    1:  (disk_interface=virtio).(guest_os=fedora)
    dep = []
    disk_interface = virtio
    guest_os = fedora
    name = (disk_interface=virtio).(guest_os=fedora)
    shortname = (disk_interface=virtio).(guest_os=fedora)
dict    2:  (disk_interface=virtio).(guest_os=ubuntu)
    dep = []
    disk_interface = virtio
    guest_os = ubuntu
    name = (disk_interface=virtio).(guest_os=ubuntu)
    shortname = (disk_interface=virtio).(guest_os=ubuntu)
dict    3:  (disk_interface=hda).(guest_os=fedora)
    dep = []
    disk_interface = hda
    guest_os = fedora
```

(continues on next page)

(continued from previous page)

```
    name = (disk_interface=hda).(guest_os=fedora)
    shortname = (disk_interface=hda).(guest_os=fedora)
dict    4:  (disk_interface=hda).(guest_os=ubuntu)
    dep = []
    disk_interface = hda
    guest_os = ubuntu
    name = (disk_interface=hda).(guest_os=ubuntu)
    shortname = (disk_interface=hda).(guest_os=ubuntu)
```

6.2.4 Dependencies

Often it is necessary to dictate relationships between variants. In this way, the order of the resulting variant sets may be influenced. This is accomplished by listing the names of all parents (in order) after the child's variant name. However, the influence of dependencies is 'weak', in that any later defined, lower-level (higher indentation) definitions, and/or filters (see section *filters*) can remove or modify dependents. For example, if testing unattended installs, each virtual machine must be booted before, and shutdown after:

```
variants:
    - one:
        key1 = Hello
    - two: one
        key2 = World
    - three: one two
```

Results in the correct sequence of variant sets: one, two, *then* three.

6.2.5 Filters

Filter statements allow modifying the resultant set of keys based on the name of the variant set (see section *variants*). Filters can be used in 3 ways: Limiting the set to include only combination names matching a pattern. Limiting the set to exclude all combination names not matching a pattern. Modifying the set or contents of key/value pairs within a matching combination name.

Names are matched by pairing a variant name component with the character(s) ',' meaning OR, '..' meaning AND, and '.' meaning IMMEDIATELY-FOLLOWED-BY. When used alone, they permit modifying the list of key/values previously defined. For example:

```
Linux..OpenSuse:
initrd = initrd
```

Modifies all variants containing 'Linux' followed anywhere thereafter with 'OpenSuse', such that the 'initrd' key is created or overwritten with the value 'initrd'.

When a filter is preceded by the keyword 'only' or 'no', it limits the selection of variant combination's This is used where a particular set of one or more variant combination's should be considered selectively or exclusively. When given an extremely large matrix of variants, the 'only' keyword is convenient to limit the result set to only those matching the filter. Whereas the 'no' keyword could be used to remove particular conflicting key/value sets under other variant combination names. For example:

```
only Linux..Fedora..64
```

Would reduce an arbitrarily large matrix to only those variants who's names contain Linux, Fedora, and 64 in them.

However, note that any of these filters may be used within named variants as well. In this application, they are only evaluated when that variant name is selected for inclusion (implicitly or explicitly) by a higher-order. For example:

```
variants:
    - one:
        key1 = Hello
variants:
    - two:
        key2 = Complicated
    - three: one two
        key3 = World
variants:
    - default:
        only three
        key2 =

only default
```

Results in the following outcome:

```
name = default.three.one
key1 = Hello
key2 =
key3 = World
```

6.2.6 Value Substitutions

Value substitution allows for selectively overriding precedence and defining part or all of a future key's value. Using a previously defined key, it's value may be substituted in or as a another key's value. The syntax is exactly the same as in the bash shell, where as a key's value is substituted in wherever that key's name appears following a '$' character. When nesting a key within other non-key-name text, the name should also be surrounded by '{', and '}' characters.

Replacement is context-sensitive, thereby if a key is redefined within the same, or, higher-order block, that value will be used for future substitutions. If a key is referenced for substitution, but hasn't yet been defined, no action is taken. In other words, the $key or ${key} string will appear literally as or within the value. Nesting of references is not supported (i.e. key substitutions within other substitutions.

For example, if `one = 1, two = 2, and three = 3`; then, `order = ${one}${two}${three}` results in `order = 123`. This is particularly handy for rooting an arbitrary complex directory tree within a predefined top-level directory.

An example of context-sensitivity,

```
key1 = default value
key2 = default value

sub = "key1: ${key1}; key2: ${key2};"

variants:
    - one:
        key1 = Hello
        sub = "key1: ${key1}; key2: ${key2};"
    - two: one
        key2 = World
        sub = "key1: ${key1}; key2: ${key2};"
    - three: one two
        sub = "key1: ${key1}; key2: ${key2};"
```

Results in the following,

```
dict    1:  one
    dep = []
    key1 = Hello
    key2 = default value
    name = one
    shortname = one
    sub = key1: Hello; key2: default value;
dict    2:  two
    dep = ['one']
    key1 = default value
    key2 = World
    name = two
    shortname = two
    sub = key1: default value; key2: World;
dict    3:  three
    dep = ['one', 'two']
    key1 = default value
    key2 = default value
    name = three
    shortname = three
    sub = key1: default value; key2: default value;
```

6.2.7 Key sub-arrays

Parameters for objects like VM's utilize array's of keys specific to a particular object instance. In this way, values specific to an object instance can be addressed. For example, a parameter 'vms' lists the VM objects names to instantiate in the current frame's test. Values specific to one of the named instances should be prefixed to the name:

```
vms = vm1 second_vm another_vm
mem = 128
mem_vm1 = 512
mem_second_vm = 1024
```

The result would be, three virtual machine objects are create. The third one (another_vm) receives the default 'mem' value of 128. The first two receive specialized values based on their name.

The order in which these statements are written in a configuration file is not important; statements addressing a single object always override statements addressing all objects. Note: This is contrary to the way the Cartesian configuration file as a whole is parsed (top-down).

6.2.8 Heterogeneous guest variants

The '`join`' filter combined with the '`suffix`' operator can be utilized together in order to produce guest variants with different guest OS or other types of configuration within the same test. Adding the '`suffix`' keyword at the end of each variant to be combined and joining all the variants in the end will produce one final variant product with all participating variant dictionaries separately suffixed and combined into one final dictionary. For instance the following definition

```
variants:
    - one:
        key1 = Hello
        key2 = Brave
```

(continues on next page)

(continued from previous page)

```
        suffix _v1
    - two:
        key1 = Bye
        key2 = Brave
        key3 = World
        suffix _v2
    - three:
variants:
    - four:
        key4 = foo
        only one
    - five:
        key4 = bar
    - six:
        key1 = foo
        key2 = bar
        key3 = baz
        only two

join one two
```

will join variants `one` and `two` with suffixed parameters `key1_v1 = Hello` and `key1_v2 = Bye` in each of the following resulting dictionaries:

```
dict    1:  four.one.five.two
    dep = []
    key1_v1 = Hello
    key1_v2 = Bye
    key2 = Brave
    key3 = World
    key4 = bar
    name = four.one.five.two
    shortname = four.one.five.two
dict    2:  four.one.six.two
    dep = []
    key1 = foo
    key1_v1 = Hello
    key1_v2 = Bye
    key2 = bar
    key2_v1 = Brave
    key2_v2 = Brave
    key3 = baz
    key3_v2 = World
    key4 = foo
    name = four.one.six.two
    shortname = four.one.six.two
dict    3:  five.one.two
    dep = []
    key1_v1 = Hello
    key1_v2 = Bye
    key2 = Brave
    key3 = World
    key4 = bar
    name = five.one.two
    shortname = five.one.two
dict    4:  five.one.six.two
```

(continues on next page)

(continued from previous page)

```
    dep = []
    key1 = foo
    key1_v1 = Hello
    key1_v2 = Bye
    key2 = bar
    key2_v1 = Brave
    key2_v2 = Brave
    key3 = baz
    key3_v2 = World
    key4 = bar
    name = five.one.six.two
    shortname = five.one.six.two
```

Regarding the choice of variants, using the `join one two` also plays the role of a filter for `three` which restricts the final selection to all joined `one-two` test variants. The `join one` operation is thus equivalent to an `only one` filter. The outcome above is additionally filtered by `only` filters, which restrict the initial joined Cartesian products:

```
dict    1:  four.one.two
dict    2:  four.one.five.two
dict    3:  four.one.six.two
dict    4:  five.one.four.two
dict    5:  five.one.two
dict    6:  five.one.six.two
dict    7:  six.one.four.two
dict    8:  six.one.five.two
dict    9:  six.one.two
```

of pairs `<four-or-five-or-six>.one` and `<four-or-five-or-six>.two` to variants containing `four.one`, `six.two`, or `five.<one-or-two>`

```
dict    1:  four.one.five.two
dict    2:  four.one.six.two
dict    3:  five.one.two
dict    4:  five.one.six.two
```

One important point to consider when using heterogeneous variants is that parameter overwriting of suffixed parameters is only possible using regex operators like '`?=`', '`?<=`', and '`?+=`'. Since adding a suffix will transform all parameters within the variant (or at least the ones that have more general nonsuffixed version of different value), regular overwriting will only match the general parameter and will not perform a search for the suffixed versions for performance reasons. Using the above operators with proper regular expression for the key will solve this.

6.2.9 Include statements

The '`include`' statement is utilized within a Cartesian configuration file to better organize related content. When parsing, the contents of any referenced files will be evaluated as soon as the parser encounters the `include` statement. The order in which files are included is relevant, and will carry through any key/value substitutions (see section *key_sub_arrays*) as if parsing a complete, flat file.

6.2.10 Combinatorial outcome

The parser is available as both a python module and command-line tool for examining the parsing results in a text-based listing. To utilize it on the command-line, run the module followed by the path of the configuration file to parse. For example, `common_lib/cartesian_config.py tests/libvirt/tests.cfg`.

The output will be just the names of the combinatorial result set items (see short-names, section Variants). However, the '`--contents`' parameter may be specified to examine the output in more depth. Internally, the key/value data is stored/accessed similar to a python dictionary instance. With the collection of dictionaries all being part of a python list-like object. Irrespective of the internals, running this module from the command-line is an excellent tool for both reviewing and learning about the Cartesian Configuration format.

In general, each individual combination of the defined variants provides the parameters for a single test. Testing proceeds in order, through each result, passing the set of keys and values through to the harness and test code. When examining Cartesian configuration files, it's helpful to consider the earliest key definitions as "defaults", then look to the end of the file for other top-level override to those values. If in doubt of where to define or set a key, placing it at the top indentation level, at the end of the file, will guarantee it is used.

6.2.11 Formal definition

- A list of dictionaries is referred to as a frame.
- The parser produces a list of dictionaries (dicts). Each dictionary contains a set of key-value pairs.
- Each dict contains at least three keys: name, shortname and depend. The values of name and shortname are strings, and the value of depend is a list of strings.
- The initial frame contains a single dict, whose name and shortname are empty strings, and whose depend is an empty list.
- Parsing dict contents
  - The dict parser operates on a frame, referred to as the current frame.
  - A statement of the form <key> = <value> sets the value of <key> to <value> in all dicts of the current frame. If a dict lacks <key>, it will be created.
  - A statement of the form <key> += <value> appends <value> to the value of <key> in all dicts of the current frame. If a dict lacks <key>, it will be created.
  - A statement of the form <key> <= <value> pre-pends <value> to the value of <key> in all dicts of the current frame. If a dict lacks <key>, it will be created.
  - A statement of the form <key> ~= <value> sets the value of <key> to <value> in all dicts of the current frame unless the <key> was already defined. Otherwise the original value is preserved.
  - A statement of the form <key> ?= <value> sets the value of <key> to <value>, in all dicts of the current frame, but only if <key> exists in the dict. The operators ?+= and ?<= are also supported.
  - A statement of the form no <regex> removes from the current frame all dicts whose name field matches <regex>.
  - A statement of the form only <regex> removes from the current frame all dicts whose name field does not match <regex>.
- Content exceptions
  - Single line exceptions have the format <regex>: <key> <operator> <value> where <operator> is any of the operators listed above (e.g. =, +=, ?<=). The statement following the regular expression <regex> will apply only to the dicts in the current frame whose name partially matches <regex> (i.e. contains a substring that matches <regex>).
  - A multi-line exception block is opened by a line of the format <regex>:. The text following this line should be indented. The statements in a multi-line exception block may be assignment statements (such as <key> = <value>) or no or only statements. Nested multi-line exceptions are allowed.
- Parsing Variants

  - A variants block is opened by a `variants:` statement. The indentation level of the statement places the following set within the outer-most context-level when nested within other `variant:` blocks. The contents of the `variants:` block must be further indented.
  - A variant-name may optionally follow the `variants` keyword, before the `:` character. That name will be inherited by and decorate all block content as the key for each variant contained in it's the block.
  - The name of the variants are specified as `- <variant_name>:`. Each name is pre-pended to the name field of each dict of the variant's frame, along with a separator dot ('.').
  - The contents of each variant may use the format `<key> <op> <value>`. They may also contain further `variants:` statements.
  - If the name of the variant is not preceeded by a @ (i.e. - @<variant_name>:), it is pre-pended to the shortname field of each dict of the variant's frame. In other words, if a variant's name is preceeded by a @, it is omitted from the shortname field.
  - Each variant in a variants block inherits a copy of the frame in which the variants: statement appears. The 'current frame', which may be modified by the dict parser, becomes this copy.
  - The frames of the variants defined in the block are joined into a single frame. The contents of frame replace the contents of the outer containing frame (if there is one).

- Filters
  - Filters can be used in 3 ways:
    * `only <filter>`
    * `no <filter>`
    * `<filter>: starts a conditional block (see section :ref:`filters_`)`
  - Syntax:

```
.. means AND
. means IMMEDIATELY-FOLLOWED-BY
```

- Example:

```
qcow2..Fedora.14, RHEL.6..raw..boot, smp2..qcow2..migrate..ide
```

```
means match all dicts whose names have:
(qcow2 AND (Fedora IMMEDIATELY-FOLLOWED-BY 14)) OR
((RHEL IMMEDIATELY-FOLLOWED-BY 6) AND raw AND boot) OR
(smp2 AND qcow2 AND migrate AND ide)
```

- Note:

```
'qcow2..Fedora.14' is equivalent to 'Fedora.14..qcow2'.
```

```
'qcow2..Fedora.14' is not equivalent to 'qcow2..14.Fedora'.
'ide, scsi' is equivalent to 'scsi, ide'.
```

6.2.12 Examples

- A single dictionary:

```
key1 = value1
key2 = value2
key3 = value3

Results in the following::

Dictionary #0:
    depend = []
    key1 = value1
    key2 = value2
    key3 = value3
    name =
    shortname =
```

- Adding a variants block:

```
key1 = value1
key2 = value2
key3 = value3

variants:
    - one:
    - two:
    - three:
```

Results in the following:

```
Dictionary #0:
    depend = []
    key1 = value1
    key2 = value2
    key3 = value3
    name = one
    shortname = one
Dictionary #1:
    depend = []
    key1 = value1
    key2 = value2
    key3 = value3
    name = two
    shortname = two
Dictionary #2:
    depend = []
    key1 = value1
    key2 = value2
    key3 = value3
    name = three
    shortname = three
```

- Modifying dictionaries inside a variant:

```
key1 = value1
key2 = value2
key3 = value3

variants:
    - one:
```

(continues on next page)

(continued from previous page)

```
        key1 = Hello World
        key2 <= some_prefix_
    - two:
        key2 <= another_prefix_
    - three:
```

Results in the following:

```
Dictionary #0:
    depend = []
    key1 = Hello World
    key2 = some_prefix_value2
    key3 = value3
    name = one
    shortname = one
Dictionary #1:
    depend = []
    key1 = value1
    key2 = another_prefix_value2
    key3 = value3
    name = two
    shortname = two
Dictionary #2:
    depend = []
    key1 = value1
    key2 = value2
    key3 = value3
    name = three
    shortname = three
```

- Adding dependencies:

```
key1 = value1
key2 = value2
key3 = value3

variants:
    - one:
        key1 = Hello World
        key2 <= some_prefix_
    - two: one
        key2 <= another_prefix_
    - three: one two
```

Results in the following:

```
Dictionary #0:
    depend = []
    key1 = Hello World
    key2 = some_prefix_value2
    key3 = value3
    name = one
    shortname = one
Dictionary #1:
    depend = ['one']
    key1 = value1
    key2 = another_prefix_value2
```

(continues on next page)

(continued from previous page)

```
    key3 = value3
    name = two
    shortname = two
Dictionary #2:
    depend = ['one', 'two']
    key1 = value1
    key2 = value2
    key3 = value3
    name = three
    shortname = three
```

- Multiple variant blocks:

```
key1 = value1
key2 = value2
key3 = value3

variants:
    - one:
        key1 = Hello World
        key2 <= some_prefix_
    - two: one
        key2 <= another_prefix_
    - three: one two

variants:
    - A:
    - B:
```

Results in the following:

```
Dictionary #0:
    depend = []
    key1 = Hello World
    key2 = some_prefix_value2
    key3 = value3
    name = A.one
    shortname = A.one
Dictionary #1:
    depend = ['A.one']
    key1 = value1
    key2 = another_prefix_value2
    key3 = value3
    name = A.two
    shortname = A.two
Dictionary #2:
    depend = ['A.one', 'A.two']
    key1 = value1
    key2 = value2
    key3 = value3
    name = A.three
    shortname = A.three
Dictionary #3:
    depend = []
    key1 = Hello World
    key2 = some_prefix_value2
    key3 = value3
```

(continues on next page)

(continued from previous page)

```
    name = B.one
    shortname = B.one
Dictionary #4:
    depend = ['B.one']
    key1 = value1
    key2 = another_prefix_value2
    key3 = value3
    name = B.two
    shortname = B.two
Dictionary #5:
    depend = ['B.one', 'B.two']
    key1 = value1
    key2 = value2
    key3 = value3
    name = B.three
    shortname = B.three
```

- Filters, `no` and `only`:

```
key1 = value1
key2 = value2
key3 = value3

variants:
    - one:
        key1 = Hello World
        key2 <= some_prefix_
    - two: one
        key2 <= another_prefix_
    - three: one two

variants:
    - A:
        no one
    - B:
        only one,three
```

Results in the following:

```
Dictionary #0:
    depend = ['A.one']
    key1 = value1
    key2 = another_prefix_value2
    key3 = value3
    name = A.two
    shortname = A.two
Dictionary #1:
    depend = ['A.one', 'A.two']
    key1 = value1
    key2 = value2
    key3 = value3
    name = A.three
    shortname = A.three
Dictionary #2:
    depend = []
    key1 = Hello World
    key2 = some_prefix_value2
```

(continues on next page)

(continued from previous page)

```
    key3 = value3
    name = B.one
    shortname = B.one
Dictionary #3:
    depend = ['B.one', 'B.two']
    key1 = value1
    key2 = value2
    key3 = value3
    name = B.three
    shortname = B.three
```

- Short-names:

```
key1 = value1
key2 = value2
key3 = value3

variants:
    - one:
        key1 = Hello World
        key2 <= some_prefix_
    - two: one
        key2 <= another_prefix_
    - three: one two

variants:
    - @A:
        no one
    - B:
        only one,three
```

Results in the following:

```
Dictionary #0:
    depend = ['A.one']
    key1 = value1
    key2 = another_prefix_value2
    key3 = value3
    name = A.two
    shortname = two
Dictionary #1:
    depend = ['A.one', 'A.two']
    key1 = value1
    key2 = value2
    key3 = value3
    name = A.three
    shortname = three
Dictionary #2:
    depend = []
    key1 = Hello World
    key2 = some_prefix_value2
    key3 = value3
    name = B.one
    shortname = B.one
Dictionary #3:
    depend = ['B.one', 'B.two']
    key1 = value1
```

(continues on next page)

(continued from previous page)

```
    key2 = value2
    key3 = value3
    name = B.three
    shortname = B.three
```

- Exceptions:

```
key1 = value1
key2 = value2
key3 = value3

variants:
    - one:
        key1 = Hello World
        key2 <= some_prefix_
    - two: one
        key2 <= another_prefix_
    - three: one two

variants:
    - @A:
        no one
    - B:
        only one,three

three: key4 = some_value

A:
    no two
    key5 = yet_another_value
```

Results in the following:

```
Dictionary #0:
    depend = ['A.one', 'A.two']
    key1 = value1
    key2 = value2
    key3 = value3
    key4 = some_value
    key5 = yet_another_value
    name = A.three
    shortname = three
Dictionary #1:
    depend = []
    key1 = Hello World
    key2 = some_prefix_value2
    key3 = value3
    name = B.one
    shortname = B.one
Dictionary #2:
    depend = ['B.one', 'B.two']
    key1 = value1
    key2 = value2
    key3 = value3
    key4 = some_value
    name = B.three
    shortname = B.three
```

6.2.13 Default Configuration Files

The test configuration files are used for controlling the framework, by specifying parameters for each test. The parser produces a list of key/value sets, each set pertaining to a single test. Variants are organized into separate files based on scope and/or applicability. For example, the definitions for guest operating systems is sourced from a shared location since all virtualization tests may utilize them.

For each set/test, keys are interpreted by the test dispatching system, the pre-processor, the test module itself, then by the post-processor. Some parameters are required by specific sections and others are optional. When required, parameters are often commented with possible values and/or their effect. There are select places in the code where in-memory keys are modified, however this practice is discouraged unless there's a very good reason.

When `avocado vt-bootstrap --vt-type [type]` is executed (see section *Bootstrapping Avocado-VT*), copies of the sample configuration files are copied for use under the `backends/[type]/cfg` subdirectory of the virtualization technology-specific directory. For example, `backends/qemu/cfg/base.cfg`.

Relative Directory or File	Description
cfg/tests.cfg	The first file read that includes all other files, then the master set of filters to select the actual test set to be run. Normally this file never needs to be modified unless precise control over the test-set is needed when utilizing the autotest-client (only).
cfg/tests-shared.cfg	Included by `tests.cfg` to indirectly reference the remaining set of files to include as well as set some global parameters. It is used to allow customization and/or insertion within the set of includes. Normally this file never needs to be modified.
cfg/base.cfg	Top-level file containing important parameters relating to all tests. All keys/values defined here will be inherited by every variant unless overridden. This is the *first* file to check for settings to change based on your environment
cfg/build.cfg	Configuration specific to pre-test code compilation where required/requested. Ignored when a client is not setup for build testing.
cfg/subtests.cfg	Automatically generated based on the test modules and test configuration files found when the `avocado vt-bootstrap` is used. Modifications are discouraged since they will be lost next time bootstrap is used.
cfg/guest-os.cfg	Automatically generated when `avocado vt-bootstrap` is used from files within `shared/cfg/guest-os/`. Defines all supported guest operating system types, architectures, installation images, parameters, and disk device or image names.
cfg/guest-hw.cfg	All virtual and physical hardware related parameters are organized within variant names. Within subtest variants or the top-level test set definition, hardware is specified by Including, excluding, or filtering variants and keys established in this file.
cfg/cdkeys.cfg	Certain operating systems require non-public information in order to operate and or install properly. For example, installation numbers and license keys. None of the values in this file are populated automatically. This file should be edited to supply this data for use by the unattended install test.
cfg/virtio-win.cfg	Paravirtualized hardware when specified for Windows testing, must have dependent drivers installed as part of the OS installation process. This file contains mandatory variants and keys for each Windows OS version, specifying the host location and installation method for each driver.

6.3 Building test applications

This is a description of how to build test applications from a test case.

6.3.1 Dependencies

If you write an application that is supposed to be run on the test-target, place it in the directory *../deps/<name>/* relative to where your test case is placed. The easiest way to obtain the full path to this directory is by calling *data_dir.get_deps_dir("<name>")*. Don't forget to add *from virttest import data_dir* to your test case.

Besides the source file, create a Makefile that will be used to build your test application. The below example shows a Makefile for the application for the timedrift test cases. The *remote_build* module requires that a Makefile is included with all test applications.

```
CFLAGS+=-Wall
LDLIBS+=-lrt

.PHONY: clean

all: clktest get_tsc

clktest: clktest.o

get_tsc: get_tsc.o

clean:
        rm -f clktest get_tsc
```

6.3.2 remote_build

To simplify the building of applications on target, and to simplify avoiding the building of applications on target when they are installed pre-built, use the *remote_build* module. This module handles both the transfer of files, and running *make* on target.

A simple example:

```
address = vm.get_address(0)
source_dir = data_dir.get_deps_dir("<testapp>")
builder = remote_build.Builder(params, address, source_dir)
full_build_path = builder.build()
```

In this case, we utilize the *.build()* method, which execute the necessary methods in *builder* to copy all files to target and run make (if needed). When done, *.build()* will return the full path on target to the application that was just built. Be sure to use this path when running your test application, as the path is changed if the parameters of the build is changed. For example:

```
session.cmd_status("%s --test" % os.path.join(full_build_path, "testapp"))
```

The *remote_build.Builder* class can give you fine-grained control over your build process as well. Another way to write the above *.build()* invocation above is:

```
builder = remote_build.Builder(params, address, source_dir)
if builder.sync_directories():
    builder.make()
full_build_path = builder.full_build_path
```

This pattern can be useful if you e.g. would like to add an additional command to run before *builder.make()*, perhaps to install some extra dependencies.

6.4 Networking

Here we have notes about networking setup in Avocado-VT.

6.4.1 Configuration

How to configure to allow all the traffic to be forwarded across the virbr0 bridge: Execute the command

```
$ echo "-I FORWARD -m physdev --physdev-is-bridged -j ACCEPT" > /etc/sysconfig/
↪iptables-forward-bridged
$ lokkit --custom-rules=ipv4:filter:/etc/sysconfig/iptables-forward-bridged
$ service libvirtd reload
```

Configure Static IP address in Avocado-VT

Sometimes, we need to test with guest(s) which have static ip address(es).

- e.g. No real/emulated DHCP server in test environment.
- e.g. Test with old image we don't want to change the net config.
- e.g. Test when DHCP exists problem.

Create a bridge (for example, 'vbr') in host, configure its ip to 192.168.100.1, guest can access host by it. And assign nic(s)' ip in tests.cfg, and execute test as usual.

tests.cfg:

```
ip_nic1 = 192.168.100.119
nic_mac_nic1 = 11:22:33:44:55:67
bridge = vbr
```

6.4.2 TestCases

Ntttcp

The Ntttcp test suite is a network performance test for windows, developed by Microsoft. It is *not* a freely redistributable binary, so you must download it from the website, here's the direct link for download (keep in mind it might change):

https://gallery.technet.microsoft.com/NTttcp-Version-528-Now-f8b12769

The knowledge base article associated with it is:

http://msdn.microsoft.com/en-us/windows/hardware/gg463264

You need to add the package to winutils.iso, the iso with utilities used to test windows. First, download the iso. *The get started documentation* can help you out with downloading if you like it, but the direct download link is here:

https://avocado-project.org/data/assets/winutils.iso

You need to put all its contents on a folder and create a new iso. Let's say you want to download the iso to `/home/kermit/Downloads/winutils.iso`. You can create the directory, go to it:

```
$ mkdir -p /home/kermit/Downloads
$ cd /home/kermit/Downloads
```

Download the iso, create 2 directories, 1 for the mount, another for the contents:

```
$ wget https://avocado-project.org/data/assets/winutils.iso
$ mkdir original
$ sudo mount -o loop winutils.iso original
$ mkdir winutils
```

Copy all contents from the original cd to the new structure:

```
$ cp -r original/* winutils/
```

Create the destination ntttcp directory on that new structure:

```
$ mkdir -p winutils/NTttcp
```

Download the installer and copy autoit script to the new structure, unmount the original mount:

```
$ cd winutils/NTttcp
$ wget https://gallery.technet.microsoft.com/NTttcp-Version-528-Now-f8b12769/file/
↪159655/1/NTttcp-v5.33.zip -O "winutils/NTttcp/NTttcp-v5.33.zip"
$ cp /usr/local/autotest/client/virt/scripts/ntttcp.au3 ./
$ sudo umount original
```

Backup the old winutils.iso and create a new winutils.iso using mkisofs:

```
$ sudo mv winutils.iso winutils.iso.bak
$ mkisofs -o winutils.iso -max-iso9660-filenames -relaxed-filenames -D --input-
↪charset iso8859-1 winutils
```

And that is it. Don't forget to keep winutils in an appropriate location that can be seen by Avocado-VT.

6.5 Performance Testing

6.5.1 Performance subtests

network

- netperf (linux and windows)
- ntttcp (windows)

block

- iozone (linux)
- iozone (windows) (iozone has its own result analysis module)
- iometer (windows) (not push upstream)
- ffsb (linux)
- qemu_iotests (host)
- fio (linux)

6.5.2 Environment setup

> Autotest already supports prepare environment for performance testing, guest/host need to be reboot for some configuration. setup script

Autotest supports to numa pining. Assign "numanode=-1" in tests.cfg, then vcpu threads/vhost_net threads/VM memory will be pined to last numa node. If you want to pin other processes to numa node, you can use numctl and taskset.

```
memory:
$ numactl -m $n $cmdline
cpu:
$ taskset $node_mask $thread_id
```

The following content is manual guide.

```
1.First level pinning would be to use numa pinning when starting the guest.
e.g.  $ numactl -c 1 -m 1 qemu-kvm  -smp 2 -m 4G <> (pinning guest memory and cpus to␣
↪numa-node 1)

2.For a single instance test, it would suggest trying a one to one mapping of vcpu to␣
↪pyhsical core.
e.g.
get guest vcpu threads id
$ taskset -p 40 $vcpus1  #(pinning vcpu1 thread to pyshical cpu #6 )
$ taskset -p 80 $vcpus2  #(pinning vcpu2 thread to physical cpu #7 )

3.To pin vhost on host. get vhost PID and then use taskset to pin it on the same␣
↪soket.
e.g
$ taskset -p 20 $vhost #(pinning vcpu2 thread to physical cpu #5 )

4.In guest,pin the IRQ to one core and the netperf to another.
1) make sure irqbalance is off - `$ service irqbalance stop`
2) find the interrupts - `$ cat /proc/interrupts`
3) find the affinity mask for the interrupt(s) - `$ cat /proc/irq/<irq#>/smp_affinity`
4) change the value to match the proper core.make sure the vlaue is cpu mask.
e.g. pin the IRQ to first core.
   $ echo 01>/proc/irq/$virti0-input/smp_affinity
   $ echo 01>/proc/irq/$virti0-output/smp_affinity
5)pin the netserver to another core.
e.g.
$ taskset -p 02 netserver

5.For host to guest scenario. to get maximum performance. make sure to run netperf on␣
↪different cores on the same numa node as the guest.
e.g.
$ numactl -m 1 netperf -T 4 (pinning netperf to physical cpu #4)
```

6.5.3 Execute testing

- Submit jobs in Autotest server, only execute netperf.guset_exhost for three times.

`tests.cfg`:

```
only netperf.guest_exhost
variants:
    - repeat1:
```

(continues on next page)

(continued from previous page)

```
    - repeat2:
    - repeat3:
# vbr0 has a static ip: 192.168.100.16
bridge=vbr0
# virbr0 is created by libvirtd, guest nic2 get ip by dhcp
bridge_nic2 = virbr0
# guest nic1 static ip
ip_nic1 = 192.168.100.21
# external host static ip:
client = 192.168.100.15
```

Result files:

```
$ cd /usr/local/autotest/results/8-debug_user/192.168.122.1/
$ find .|grep RHS
kvm.repeat1.r61.virtio_blk.smp2.virtio_net.RHEL.6.1.x86_64.netperf.exhost_guest/
↪results/netperf-result.RHS
kvm.repeat2.r61.virtio_blk.smp2.virtio_net.RHEL.6.1.x86_64.netperf.exhost_guest/
↪results/netperf-result.RHS
kvm.repeat3.r61.virtio_blk.smp2.virtio_net.RHEL.6.1.x86_64.netperf.exhost_guest/
↪results/netperf-result.RHS
```

- Submit same job in another env (different packages) with same configuration

Result files:

```
$ cd /usr/local/autotest/results/9-debug_user/192.168.122.1/
$ find .|grep RHS
kvm.repeat1.r61.virtio_blk.smp2.virtio_net.RHEL.6.1.x86_64.netperf.exhost_guest/
↪results/netperf-result.RHS
kvm.repeat2.r61.virtio_blk.smp2.virtio_net.RHEL.6.1.x86_64.netperf.exhost_guest/
↪results/netperf-result.RHS
kvm.repeat3.r61.virtio_blk.smp2.virtio_net.RHEL.6.1.x86_64.netperf.exhost_guest/
↪results/netperf-result.RHS
```

6.5.4 Analysis result

Config file: perf.conf

```
[ntttcp]
result_file_pattern = .*.RHS
ignore_col = 1
avg_update =

[netperf]
result_file_pattern = .*.RHS
ignore_col = 2
avg_update = 4,2,3|14,5,12|15,6,13

[iozone]
result_file_pattern =
```

Execute regression.py to compare two results:

```
login autotest server
$ cd /usr/local/autotest/client/tools
$ python regression.py netperf /usr/local/autotest/results/8-debug_user/192.168.122.1/
↪ /usr/local/autotest/results/9-debug_user/192.168.122.1/
```

T-test:

- scipy: http://www.scipy.org/
- t-test: http://en.wikipedia.org/wiki/Student's_t-test
- Two python modules (scipy and numpy) are needed.
- Script to install numpy/scipy on rhel6 automatically: https://github.com/kongove/misc/blob/master/scripts/install-numpy-scipy.sh

Unpaired T-test is used to compare two samples, user can check p-value to know if regression bug exists. If the difference of two samples is considered to be not statistically significant(p <= 0.05), it will add a '+' or '-' before p-value. ('+': avg_sample1 < avg_sample2, '-': avg_sample1 > avg_sample2)

- - only over 95% confidence results will be added "+/-" in "Significance" part.
- + for cpu-usage means regression, "+" for throughput means improvement."

6.5.5 Regression results

- Every Avg line represents the average value based on *$n* repetitions of the same test, and the following SD line represents the Standard Deviation between the *$n* repetitions.
- The Standard deviation is displayed as a percentage of the average.
- The significance of the differences between the two averages is calculated using unpaired T-test that takes into account the SD of the averages.
- The paired t-test is computed for the averages of same category.
- only over 95% confidence results will be added "+/-" in "Significance" part. "+" for cpu-usage means regression, "+" for throughput means improvement.

Highlight HTML result

- green/red –> good/bad
- Significance is larger than 0.95 –> green
- dark green/red –> important (eg: cpu)
- light green/red –> other
- test time
- version (only when diff)
- other: repeat time, title
- user light green/red to highlight small (< %5) DIFF
- highlight Significance with same color in one raw
- add doc link to result file, and describe color in doc

netperf.avg.html - Raw data that the averages are based on.

6.6 Setup a virtual environment for multi host tests

6.6.1 Problem:

For multi-host tests multiple physical systems are often required. It is possible to use two virtual guests for most of autotest tests except for Avocado-VTs (kvm, libvirt, . . .).

However, It is possible to use Nested Virtualization to serve as first level (L0) guests and run nested guests inside them.

This page explains, how to setup (Fedora/RHEL) and use single computer with nested virtualization for such cases. Be careful that nested virtualization works usually right, but there are cases where it might lead to hidden problems. Do not use nested virtualization for production testing.

6.6.2 Nested Virtualization:

1. **Emulated:**
    - qemu very slow
2. **Hardware accelerated:**
    - Hardware for the accelerated nested virtualization AMD Phenom and never core extension (smv, NPT) Intel Nehalem and never core extension (vmx, EPT)
    - Software which supports the accelerated nested virtualization kvm, xen, vmware, ... almost the same speed like native guest (1.0-0.1 of native quest performance). Performance depends on the type of load. IO load could be quite slow. Without vt-d or AMD-Vi and network device pass through.

6.6.3 Configuration for multi-host virt tests:

6.6.4 Config of host system

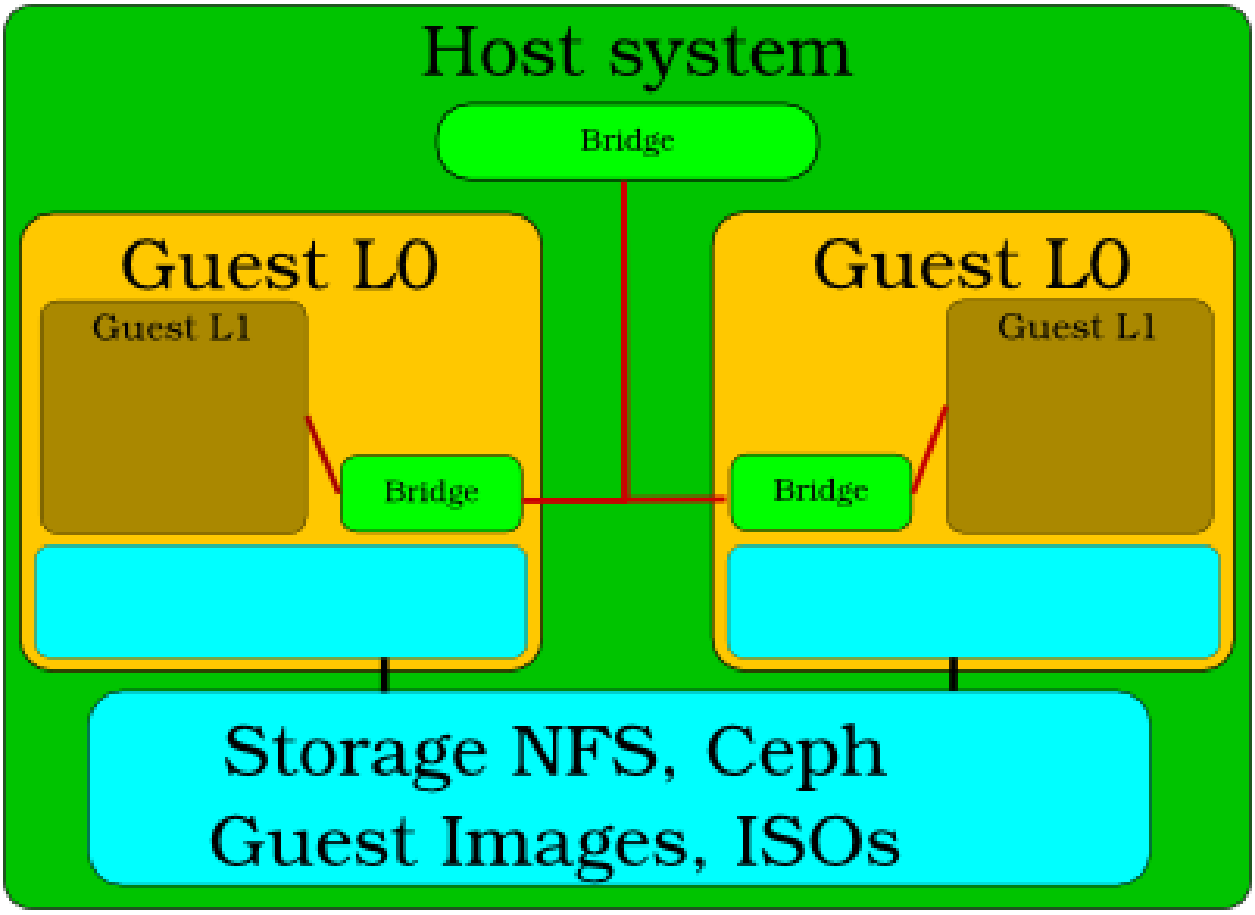
- Intel CPU

  `options kvm_intel nested=1` to the end of some modules config file /etc/modprobe.d/modules.conf
- AMD CPU

  `options kvm_amd nested=1` to the end of some modules config file /etc/modprobe.d/modules.conf

6.6.5 Config of Guest L0

- **Intel CPU**
    - **Virtual manager** `Procesor config->CPU Features->vmx set to require`
    - **Native qemu-kvm** `qemu-kvm -cpu core2duo,+vmx -enable-kvm .....`
- **AMD CPU**
    - **Virtual manager** `Procesor config->CPU Features->svm set to require`
    - **Native qemu-kvm** `qemu-kvm -cpu qemu64,+svm -enable-kvm .....`


Host system
Bridge
Guest L0
Guest L1
Bridge
Guest L0
Guest L1
Bridge
Storage NFS, Ceph
Guest Images, ISOs

6.6.6 Config of Guest L0 System

Connect to host bridge with guest L0 bridge without DHCP (dhcp collision with host system dhcp).

1. Destroy libvirt bridge which contain dhcp.
2. Enable network service systemctl enable network.service
3. Add new bridge `virbr0` manually and insert to them L0 network adapter `eth0` which is connected to host bridge

```
# interface connected to host system bridge
$ vi /etc/sysconfig/network-scripts/ifcfg-eth0
     NM_CONTROLLED="no"
     DEVICE="eth0"
     ONBOOT="yes"
     BRIDGE=virbr0

# Bridge has name virbr0 for compatibility with standard autotest settings.
$ vi /etc/sysconfig/network-scripts/ifcfg-virbr0
    DHCP_HOSTNAME="atest-guest"
    NM_CONTROLLED="no"
    BOOTPROTO="dhcp"
    ONBOOT="yes"
    IPV6INIT="no"
    DEVICE=virbr0
    TYPE=Bridge
    DELAY=0
```

and for sure disable NetworkManager `systemctl disable NetworkManager.service`

6.6.7 Check Guest L0 System

`modprobe kvm-intel` or `modprobe kvm-amd` should work

6.6.8 Start for multi-host virt tests:

6.6.9 Manually from host machine

```
$ cd autotest/client/tests/virt/qemu/
$ sudo rm -rf results.*; sudo ../../../../server/autoserv -m guestL0_1,guestL0_2␣
↪multi_host.srv
```

6.6.10 More details:

Set up/configuration, the root directory (of this git repo) also has simple scripts to create L1 and L2 guests. And reference of L1, L2 libvirt files are also added – https://github.com/kashyapc/nvmx-haswell/blob/master/SETUP-nVMX.rst

6.7 Multi Host Migration Tests

6.7.1 Running Multi Host Migration Tests

Avocado-VT is our test suite, but for simplicity purposes it can only run on a single host. For multi host tests, you'll need the full autotest + Avocado-VT package, and the procedure is more complex. We'll try to keep this procedure as objective as possible.

6.7.2 Prerequesites

This guide assumes that:

1) You have at least 2 virt capable machines that have shared storage setup in [insert specific path]. Let's call them `host1.foo.com` and `host2.foo.com`.

2) You can ssh into both of those machines without a password (which means there is an SSH key setup with the account you're going to use to run the tests) as root.

3) The machines should be able to communicate freely, so beware of the potential firewall complications. On each of those machines you need a specific NFS mount setup:

- /var/lib/virt_test/isos
- /var/lib/virt_test/steps_data
- /var/lib/virt_test/gpg

They all need to be backed by an NFS share read only. Why read only? Because it is safer, we exclude the chance to delete this important data by accident. Besides the data above is only needed in a read only fashion. fstab example:

```
myserver.foo.com:/virt-test/iso /var/lib/virt_test/isos nfs ro,nosuid,nodev,noatime,
↪intr,hard,tcp 0 0
myserver.foo.com:/virt-test/steps_data  /var/lib/virt_test/steps_data nfs rw,nosuid,
↪nodev,noatime,intr,hard,tcp 0 0
myserver.foo.com:/virt-test/gpg  /var/lib/virt_test/gpg nfs rw,nosuid,nodev,noatime,
↪intr,hard,tcp 0 0
```

- /var/lib/virt_test/images
- /var/lib/virt_test/images_archive

Those all need to be backed by an NFS share read write (or any other shared storage you might have). This is necessary because both hosts need to see the same coherent storage. fstab example:

```
myserver.foo.com:/virt-test/images_archive  /var/lib/virt_test/images_archive nfs rw,
↪nosuid,nodev,noatime,intr,hard,tcp 0 0
myserver.foo.com:/virt-test/images /var/lib/virt_test/images  nfs rw,nosuid,nodev,
↪noatime,intr,hard,tcp 0 0
```

The images dir must be populated with the installed guests you want to run your tests on. They must match the file names used by guest OS in Avocado-VT. For example, for RHEL 6.4, the image name Avocado-VT uses is:

```
rhel64-64.qcow2
```

double check your files are there:

```
$ ls /var/lib/virt_test/images
$ rhel64-64.qcow2
```

6.7.3 Setup step by step

First, clone the autotest repo recursively. It's a repo with lots of submodules, so you'll see a lot of output:

```
$ git clone --recursive https://github.com/autotest/autotest.git
... lots of output ...
```

Then, edit the global_config.ini file, and change the key:

```
serve_packages_from_autoserv: True
```

to:

```
serve_packages_from_autoserv: False
```

Then you need to update Avocado-VT's config files and sub tests (that live in separate repositories that are not git submodules). You don't need to download the JeOS file in this step, so simply answer 'n' to the quest

Note: The bootstrap procedure described below will be performed automatically upon running the autoserv command that triggers the test. The problem is that then you will not be able to see the config files and modify filters prior to actually running the test. Therefore this documentation will instruct you to run the steps below manually.

```
16:11:14 INFO | qemu test config helper
16:11:14 INFO |
16:11:14 INFO | 1 - Updating all test providers
16:11:14 INFO | Fetching git [REP 'git://github.com/autotest/tp-qemu.git' BRANCH
↪'master'] -> /var/tmp/autotest/client/tests/virt/test-providers.d/downloads/io-
↪github-autotest-qemu
16:11:17 INFO | git commit ID is 6046958afa1ccab7f22bb1a1a73347d9c6ed3211 (no tag␣
↪found)
16:11:17 INFO | Fetching git [REP 'git://github.com/autotest/tp-libvirt.git' BRANCH
↪'master'] -> /var/tmp/autotest/client/tests/virt/test-providers.d/downloads/io-
↪github-autotest-libvirt
16:11:19 INFO | git commit ID is edc07c0c4346f9029930b062c573ff6f5433bc53 (no tag␣
↪found)
16:11:20 INFO |
16:11:20 INFO | 2 - Checking the mandatory programs and headers
16:11:20 INFO | /usr/bin/xz
16:11:20 INFO | /usr/sbin/tcpdump
16:11:20 INFO | /usr/bin/nc
16:11:20 INFO | /sbin/ip
16:11:20 INFO | /sbin/arping
16:11:20 INFO | /usr/bin/gcc
16:11:20 INFO | /usr/include/bits/unistd.h
16:11:20 INFO | /usr/include/bits/socket.h
16:11:20 INFO | /usr/include/bits/types.h
16:11:20 INFO | /usr/include/python2.6/Python.h
16:11:20 INFO |
16:11:20 INFO | 3 - Checking the recommended programs
16:11:20 INFO | Recommended command missing. You may want to install it if not␣
↪building it from source. Aliases searched: ('qemu-kvm', 'kvm')
16:11:20 INFO | Recommended command qemu-img missing. You may want to install it if␣
↪not building from source.
16:11:20 INFO | Recommended command qemu-io missing. You may want to install it if␣
↪not building from source.
16:11:20 INFO |
16:11:20 INFO | 4 - Verifying directories
16:11:20 INFO |
```

(continues on next page)

(continued from previous page)

```
16:11:20 INFO | 5 - Generating config set
16:11:20 INFO |
16:11:20 INFO | 6 - Verifying (and possibly downloading) guest image
16:11:20 INFO | File JeOS 19 x86_64 not present. Do you want to download it? (y/n) n
16:11:30 INFO |
16:11:30 INFO | 7 - Checking for modules kvm, kvm-amd
16:11:30 WARNI| Module kvm is not loaded. You might want to load it
16:11:30 WARNI| Module kvm-amd is not loaded. You might want to load it
16:11:30 INFO |
16:11:30 INFO | 8 - If you wish, take a look at the online docs for more info
16:11:30 INFO |
16:11:30 INFO | https://github.com/autotest/virt-test/wiki/GetStarted
```

Then you need to copy the multihost config file to the appropriate place:

```
cp client/tests/virt/test-providers.d/downloads/io-github-autotest-qemu/qemu/cfg/
↪multi-host-tests.cfg client/tests/virt/backends/qemu/cfg/
```

Now, edit the file:

```
server/tests/multihost_migration/control.srv
```

In there, you have to change the EXTRA_PARAMS to restrict the number of guests you want to run the tests on. On this example, we're going to restrict our tests to RHEL 6.4. The particular section of the control file should look like:

```
EXTRA_PARAMS = """
only RHEL.6.4.x86_64
"""
```

It is important to stress that the guests must be installed for this to work smoothly. Then the last step would be to run the tests. Using the same convention for the machine hostnames, here's the command you should use:

```
$ server/autotest-remote -m host1.foo.com,host2.foo.com server/tests/multihost_
↪migration/control.srv
```

Now, you'll see a boatload of output from the autotest remote output. This is normal, and you should be patient until all the tests are done.

Writing Multi Host Migration tests

Scheme:

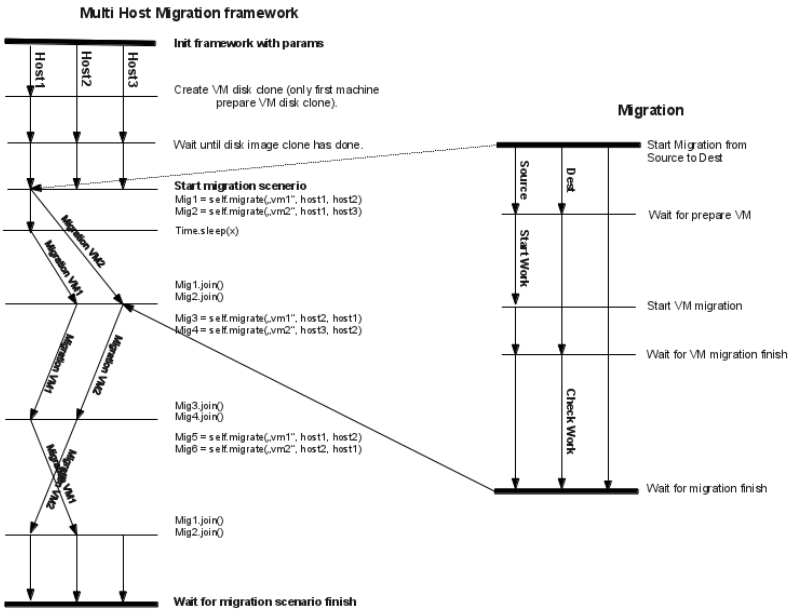
```
Source file for the diagram above (LibreOffice file)
```

Example:

```
class TestMultihostMigration(virt_utils.MultihostMigration):
    def __init__(self, test, params, env):
        super(testMultihostMigration, self).__init__(test, params, env)

    def migration_scenario(self):
        srchost = self.params.get("hosts")[0]
        dsthost = self.params.get("hosts")[1]
```

(continues on next page)



(continued from previous page)

```
    def worker(mig_data):
        vm = env.get_vm("vm1")
        session = vm.wait_for_login(timeout=self.login_timeout)
        session.sendline("nohup dd if=/dev/zero of=/dev/null &")
        session.cmd("killall -0 dd")

    def check_worker(mig_data):
        vm = env.get_vm("vm1")
        session = vm.wait_for_login(timeout=self.login_timeout)
        session.cmd("killall -9 dd")

    # Almost synchronized migration, waiting to end it.
    # Work is started only on first VM.

    self.migrate_wait(["vm1", "vm2"], srchost, dsthost,
                      worker, check_worker)

    # Migration started in different threads.
    # It allows to start multiple migrations simultaneously.

    # Starts one migration without synchronization with work.
    mig1 = self.migrate(["vm1"], srchost, dsthost,
                        worker, check_worker)

    time.sleep(20)

    # Starts another test simultaneously.
    mig2 = self.migrate(["vm2"], srchost, dsthost)
    # Wait for mig2 finish.
    mig2.join()
```

(continues on next page)

(continued from previous page)

```
        mig1.join()

mig = TestMultihostMigration(test, params, env)
# Start test.
mig.run()
```

When you call:

```
mig = TestMultihostMigration(test, params, env)
```

What happens is

1. VM's disks will be prepared.
2. The synchronization server will be started.
3. All hosts will be synchronized after VM create disks.

When you call the method:

```
migrate()
```

What happens in a diagram is:

source	destination
It prepare VM if machine is not started.	
Start work on VM.	
`mig.migrate_vms_src()`	`mig.migrate_vms_dest()`
	Check work on VM after migration.
Wait for finish migration on all hosts.	

It's important to note that the migrations are made using the `tcp` protocol, since the others don't support multi host migration.

```
def migrate_vms_src(self, mig_data):
    vm = mig_data.vms[0]
    logging.info("Start migrating now...")
    vm.migrate(mig_data.dst, mig_data.vm_ports)
```

This example migrates only the first machine defined in migration. Better example is in `virt_utils.MultihostMigration.migrate_vms_src`. This function migrates all machines defined for migration.

6.8 Links with downloadable images for virt tests

This is a central location that we aim to keep up to date with locations of iso files that might be needed for testing.

Update: Now we have a central location to define such downloads. In the source tree:

```
shared/downloads/
```

Contains a bunch of .ini files, each one with download definitions. It is expected that this will be more up to date than this page. You can see the available downloads and download the files using:

```
scripts/download_manager.py
```

6.8.1 Winutils ISO

The windows utils file can be currently found at:

https://avocado-project.org/data/assets/winutils.iso

How to update *winutils.iso*

That's basically a collection of files useful for windows testing. If you want to update that file, you'll have to pick that iso file, extract it to a directory, make changes, remaster the iso and upload back to the main location.

6.8.2 Windows virtio drivers ISO

In Avocado we mainly use fedora/RHEL based windows virtio drivers ISO that can be obtained from:

https://docs.fedoraproject.org/en-US/quick-docs/creating-windows-virtual-machines-using-virtio-drivers/index.html

6.8.3 JeOS image

You can find the JeOS images currently available here:

https://avocado-project.org/data/assets/jeos-23-64.qcow2.xz

https://avocado-project.org/data/assets/SHA1SUM_JEOS23_XZ

You can browse through https://avocado-project.org/data/assets/jeos/ and find other JeOS versions available.

How to update JeOS

The JeOS can be updated by installing it, just like a normal OS. You can do that for example with `avocado-vt`, selecting an unattended install test. In this example, we're going to use the unattended install using https kickstart and network install:

```
$ avocado run io-github-autotest-qemu.unattended_install.url.http_ks.default_install.
↪aio_native
```

The JeOS kickstart has a trick to fill the qcow2 image with zeros, so that we can squeeze these zeros later with qemu img. Once the image is installed, you can use our helper script, located at `scripts/package_jeos.py` in the avocado-vt source tree. That script uses qemu-img to trim the zeros of the image, ensuring that the resulting qcow2 image is the smallest possible. The command is similar to:

```
$ qemu-img convert -f qcow2 -O qcow2 jeos-file-backup.qcow2 jeos-file.qcow2
```

Then it'll compress it using xz, to save space and speed up downloads for `avocado-vt` users. The command is similar to:

```
$ xz jeos-file.qcow2.xz jeos-file.qcow2
```

As mentioned, the script is supposed to help you with the process.

6.8.4 Custom image

If you want to use a custom image, please keep the following in mind:

1. Per default configuration, the root account in the guests has a specific password. For example, Linux guests expect the password as defined in *virttest/shared/cfg/guest-os/Linux.cfg*.
2. The guest user's console configuration can have impact on the test code. We recommend to set *TERM=dumb* in case of *bash*. This suppresses control characters.

6.9 GlusterFS support

GlusterFS is an open source, distributed file system capable of scaling to several petabytes (actually, 72 brontobytes!) and handling thousands of clients. GlusterFS clusters together storage building blocks over Infiniband RDMA or TCP/IP interconnect, aggregating disk and memory resources and managing data in a single global namespace. GlusterFS is based on a stackable user space design and can deliver exceptional performance for diverse workloads.

More details of GlusterFS can be found under

https://www.gluster.org/

GlusterFS is added as a new block backend for qemu and to make use of this feature we require the following components.

More details of GlusterFS-QEMU Integration can be found under

http://raobharata.wordpress.com/2012/10/29/qemu-glusterfs-native-integration/

1. Qemu- 1.3, 03Dec2012
2. GlusterFS-3.4
3. Libvirt-1.0.1, 15Dec2012

6.9.1 How to use in Avocado-VT

You can use Avocado-VT to test GlusterFS support with following steps.

1) Edit qemu/cfg/tests.cfg with following changes,

```
only glusterfs_support
remove 'only no_glusterfs_support' line from the file
```

2) Optionally, edit shared/cfg/guest-hw.cfg for the gluster volume name and brick path, default is going to be,

```
gluster_volume_name = test-vol
gluster_brick = /tmp/gluster
```

6.9.2 How to use manually

The following is just an example to show how we create gluster volume and run a guest on that volume manually.

6.9.3 Starting Gluster daemon

```
$ service glusterd start
```

6.9.4 Gluster volume creation

```
$ gluster volume create [volume-name]  [hostname/host_ip]:/[brick_path]
```

E.g.: *gluster volume create test-vol satheesh.ibm.com://home/satheesh/images_gluster*

6.9.5 Qemu Img creation

```
$ qemu-img create gluster://[hostname]:0/[volume-name]/[image-name] [size]
```

E.g.: *qemu-img create gluster://satheesh.ibm.com:0/test-vol/test_gluster.img 10G*

6.9.6 Example of qemu cmd Line

```
$ qemu-system-x86_64 --enable-kvm -smp 4 -m 2048 -drive file=gluster://satheesh.ibm.
↪com/test-vol/test_gluster.img,if=virtio -net nic,macaddr=52:54:00:09:0a:0b -net tap,
↪script=/path/to/qemu-ifupVirsh
```

6.10 Setting up a Regression Test Farm for KVM

You have all upstream code, and you're wondering if the internal Red Hat testing of KVM has a lot of internal 'secret sauce'. No, it does not.

However, it is a complex endeavor, since there are *lots* of details involved. The farm setup and maintenance is not easy, given the large amounts of things that can fail (machines misbehave, network problems, git repos unavailable, so on and so forth). *You have been warned*.

With all that said, we'll share what we have been doing. We did clean up our config files and extensions and released them upstream, together with this procedure, that we hope it will be useful to you guys. Also, this will cover KVM testing on a single host, as tests involving multiple hosts and Libvirt testing are a work in progress.

The basic steps are:

1) Install an autotest server.

2) Add machines to the server (test nodes). Those machines are the virt hosts that will be tested.

3) Prepare the virt test jobs and schedule them.

4) Set up cobbler in your environment so you can install hosts.

5) Lots of trial and error until you get all little details sorted out.

We took years repeating all the steps above and perfecting the process, and we are willing to document it all to the best extent possible. I'm afraid however, that you'll have to do your homework and adapt the procedure to your environment.

6.10.1 Some conventions

We are assuming you will install autotest to its default upstream location

/usr/local/autotest

Therefore a lot of paths referred here will have this as the base dir.

6.10.2 CLI vs Web UI

During this text, we'll use frequently the terms CLI and Web UI.

By CLI we mean specifically the program:

/usr/local/autotest/cli/autotest-rpc-client

That is located in the autotest code checkout.

By Web UI, we mean the web interface of autotest, that can be accessed through

http://your-autotest-server.com/afe

6.10.3 Step 1 - Install an autotest server

Provided that you have internet on your test lab, this should be the easiest step. Pick up either a VM accessible in your lab, or a bare metal machine (it really doesn't make a difference, we use a VM here). We'll refer it from now on as the "Server" box.

The hard drive of the Server should hold enough room for test results. We found out that at least 250 GB holds data for more than 6 months, provided that QEMU doesn't crash a lot.

You'll follow the procedure described on

https://github.com/autotest/autotest/wiki/AutotestServerInstallRedHat

for Red Hat derivatives (such as Fedora and RHEL), and

https://github.com/autotest/autotest/wiki/AutotestServerInstall

for Debian derivatives (Debian, Ubuntu).

Note that using the install script referred right in the beginning of the documentation is the preferred method, and should work pretty well if you have internet on your lab. In case you don't have internet there, you'd need to follow the instructions after the 'installing with the script' instructions. Let us know if you have any problems.

6.10.4 Step 2 - Add test machines

It should go without saying, but the machines you have to add have to be virtualization capable (support KVM).

You can add machines either by using the CLI or the Web UI, following the documentation:

https://github.com/autotest/autotest/wiki/ConfiguringHosts

If you don't want to read that, I'll try to write a quick howto.

Say you have two x86_64 hosts, one AMD and the other, Intel. Their hostnames are:

foo-amd.bazcorp.com foo-intel.bazcorp.com

I would create 2 labels, amd64 and intel64, I would also create a label to indicate the machines can be provisioned by cobbler. This is because you can tell autotest to run a job in any machine that matches a given label.

Logged as the autotest user:

```
$ /usr/local/autotest/cli/autotest-rpc-client label create -t amd64
Created label:
    'amd64'
$ /usr/local/autotest/cli/autotest-rpc-client label create -t intel64
Created label:
    'intel64'
$ /usr/local/autotest/cli/autotest-rpc-client label create hostprovisioning
Created label:
    'hostprovisioning'
```

Then I'd create each machine with the appropriate labels

```
$ /usr/local/autotest/cli/autotest-rpc-client host create -t amd64 -b␣
↪hostprovisioning foo-amd.bazcorp.com
Added host:
    foo-amd.bazcorp.com

$ /usr/local/autotest/cli/autotest-rpc-client host create -t intel64 -b␣
↪hostprovisioning foo-intel.bazcorp.com
Added host:
    foo-intel.bazcorp.com
```

6.10.5 Step 3 - Prepare the test jobs

Now you have to copy the plugin we have developed to extend the CLI to parse additional information for the virt jobs:

```
$ cp /usr/local/autotest/contrib/virt/site_job.py /usr/local/autotest/cli/
```

This should be enough to enable all the extra functionality.

You also need to copy the site-config.cfg file that we published as a reference, to the qemu config module:

```
$ cp /usr/local/autotest/contrib/virt/site-config.cfg /usr/local/autotest/client/
↪tests/virt/qemu/cfg
```

Be aware that you *need* to read this file well, and later, configure it to your testing needs. We specially stress that you might want to create private git mirrors of the git repos you want to test, so you tax the upstream mirrors less, and have increased reliability.

Right now it is able to run regression testing on Fedora 18, and upstream kvm, provided that you have a cobbler instance functional, with a profile called f18-autotest-kvm that can be properly installed on your machines. Having that properly set up may open another can of worms.

One simple way to schedule the jobs, that we does use at our server, is to use cron to schedule daily testing jobs of the things you want to test. Here is an example that should work 'out of the box'. Provided that you have an internal mailing list that you created with the purpose of receiving email notifications, called autotest-virt-jobs@foocorp.com, you can stick that on the crontab of the user autotest in the Server:

```
07 00 * * 1-7 /usr/local/autotest/cli/autotest-rpc-client job create -B never -a␣
↪never -s -e autotest-virt-jobs@foocorp.com -f "/usr/local/autotest/contrib/virt/
↪control.template" -T --timestamp -m '1*hostprovisioning' -x 'only qemu-git..sanity'
↪"Upstream qemu.git sanity"
15 00 * * 1-7 /usr/local/autotest/cli/autotest-rpc-client job create -B never -a␣
↪never -s -e autotest-virt-jobs@foocorp.com -f "/usr/local/autotest/contrib/virt/
↪control.template" -T --timestamp -m '1*hostprovisioning' -x 'only f18.*' "
↪"Fedora 18 koji sanity"
```

(continues on next page)

(continued from previous page)

```
07 01 * * 1-7 /usr/local/autotest/cli/autotest-rpc-client job create -B never -a␣
↪never -s -e autotest-virt-jobs@foocorp.com -f "/usr/local/autotest/contrib/virt/
↪control.template" -T --timestamp -m '1*hostprovisioning' -x 'only qemu-git..stable'
↪"Upstream qemu.git stable"
15 01 * * 1-7 /usr/local/autotest/cli/autotest-rpc-client job create -B never -a␣
↪never -s -e autotest-virt-jobs@foocorp.com -f "/usr/local/autotest/contrib/virt/
↪control.template" -T --timestamp -m '1*hostprovisioning' -x 'only f18..stable'
↪"Fedora 18 koji stable"
```

That should be enough to have one sanity and stable job for:

- Fedora 18.
- qemu.git userspace and kvm.git kernel.

What does these 'stable' and 'sanity' jobs do? In short:

- Host OS (Fedora 18) installation through cobbler
- Latest kernel for the Host OS installation (either the last kernel update build for fedora, or check out, compile and install kvm.git).

6.10.6 sanity job

- Install latest Fedora 18 qemu-kvm, or compiles the latest qemu.git
- Installs a VM with Fedora 18, boots, reboots, does simple, single host migration with all supported protocols
- Takes about two hours. In fact, internally we test more guests, but they are not widely available (RHEL 6 and Windows 7), so we just replaced them with Fedora 18.

6.10.7 stable job

- Same as above, but many more networking, timedrift and other tests

6.10.8 Setup cobbler to install hosts

Cobbler is an installation server, that control DHCP and/or PXE boot for your x86_64 bare metal virtualization hosts. You can learn how to set it up in the following resource:

https://github.com/cobbler/cobbler/wiki

You will set it up for simple installations, and you probably just need to import a Fedora 18 DVD into it, so it can be used to install your hosts. Following the import procedure, you'll have a 'profile' created, which is a label that describes an OS that can be installed on your virtualization host. The label we chose, as already mentioned is f18-autotest-kvm. If you want to change that name, you'll have to change site-config.cfg accordingly.

Also, you will have to add your test machines to your cobbler server, and will have to set up remote control (power on/off) for them.

The following is important:

The hostname of your machine in the autotest server has to be the name of your system in cobbler.

So, for the hypothetical example you'll have to have set up systems with names foo-amd.bazcorp.com foo-intel.bazcorp.com in cobbler. That's right, the 'name' of the system has to be the 'hostname'. Otherwise, autotest will ask cobbler and cobbler will not know which machine autotest is taking about.

Other assumptions we have here:

1) We have a (read only, to avoid people deleting isos by mistake) NFS share that has the Fedora 18 DVD and other ISOS. The structure for the base dir could look something like:

```
.
|-- linux
|   `-- Fedora-18-x86_64-DVD.iso
`-- windows
    |-- en_windows_7_ultimate_x64_dvd_x15-65922.iso
    |-- virtio-win.iso
    `-- winutils.iso
```

This is just in case you are legally entitled to download and use Windows 7, for example.

2) We have another NFS share with space for backups of qcow2 images that got corrupted during testing, and you want people to analyze them. The structure would be:

```
.
|-- foo-amd
`-- bar-amd
```

That is, one directory for each host machine you have on your grid. Make sure they end up being properly configured in the kickstart.

Now here is one excerpt of kickstart with some of the gotchas we learned with experience. Some notes:

- This is not a fully formed, functional kickstart, just in case you didn't notice.
- This is provided in the hopes you read it, understand it and adapt things to your needs. If you paste this into your kickstart and tell me it doesn't work, I WILL silently ignore your email, and if you insist, your emails will be filtered out and go to the trash can.

```
install

text
reboot
lang en_US
keyboard us
rootpw --iscrypted [your-password]
firewall --disabled
selinux --disabled
timezone --utc America/New_York
firstboot --disable
services --enabled network --disabled NetworkManager
bootloader --location=mbr
ignoredisk --only-use=sda
zerombr
clearpart --all --drives=sda --initlabel
autopart
network --bootproto=dhcp --device=eth0 --onboot=on

%packages
@core
@development-libs
@development-tools
@virtualization
wget
dnsmasq
```

(continues on next page)

(continued from previous page)

```
genisoimage
python-imaging
qemu-kvm-tools
gdb
iasl
libvirt
ntpdate
gstreamer-plugins-good
gstreamer-python
dmidecode
popt-devel
libblkid-devel
pixman-devel
mtools
koji
tcpdump
bridge-utils
dosfstools
%end

%post

echo "[nfs-server-that-holds-iso-images]:[nfs-server-that-holds-iso-images]/base_path/
↪iso /var/lib/virt_test/isos nfs ro,defaults 0 0" >> /etc/fstab
echo "[nfs-server-that-holds-iso-images]:[nfs-server-that-holds-iso-images]/base_path/
↪steps_data  /var/lib/virt_test/steps_data nfs rw,nosuid,nodev,noatime,intr,hard,tcp␣
↪0 0" >> /etc/fstab
echo "[nfs-server-that-has-lots-of-space-for-backups]:/base_path/[dir-that-holds-this-
↪hostname-backups] /var/lib/virt_test/images_archive nfs rw,nosuid,nodev,noatime,
↪intr,hard,tcp 0 0" >> /etc/fstab
mkdir -p /var/lib/virt_test/isos
mkdir -p /var/lib/virt_test/steps_data
mkdir -p /var/lib/virt_test/images
mkdir -p /var/lib/virt_test/images_archive

mkdir --mode=700 /root/.ssh
echo 'ssh-dss [the-ssh-key-of-the-Server-autotest-user]' >> /root/.ssh/authorized_keys
chmod 600 /root/.ssh/authorized_keys

ntpdate [your-ntp-server]
hwclock --systohc

systemctl mask tmp.mount
%end
```

6.10.9 Painful trial and error process to adjust details

After all that, you can start running your test jobs and see what things will need to be fixed. You can run your jobs easily by logging into your Server, with the autotest user, and use the command:

```
$ /usr/local/autotest/cli/autotest-rpc-client job create -B never -a never -s -e␣
↪autotest-virt-jobs@foocorp.com -f "/usr/local/autotest/contrib/virt/control.template
↪" -T --timestamp -m '1*hostprovisioning' -x 'only f18..sanity' "Fedora 18 koji␣
↪sanity"
```

As you might have guessed, this will schedule a Fedora 18 sanity job. So go through it and fix things step by step. If

anything, you can take a look at this:

https://github.com/autotest/autotest/wiki/DiagnosingFailures

And see if it helps. You can also ask on the mailing list, but *please*, *pretty please* do your homework before you ask us to guide you through all the process step by step. This is already a step by step procedure.

All right, good luck, and happy testing!

6.11 Installing Windows virtio drivers with Avocado-VT

So, you want to use Avocado-VT to install windows guests. You also want them to be installed with the paravirtualized drivers developed for windows. You have come to the right place.

6.11.1 A bit of context on windows virtio drivers install

This method of install so far covers the storage (viostor) and network (NetKVM) drivers. Avocado-VT uses a boot floppy with a Windows answer file in order to perform unattended install of windows guests. For winXP and win2003, the unattended files are simple .ini files, while for win2008 and later, the unattended files are XML files.

In order to install the virtio drivers during guest install, KVM autotest has to inform the windows install programs *where* to find the drivers. So, we work from the following assumptions:

1. You already have an iso file that contains windows virtio drivers (inf files) for both netkvm and viostor. If you are unsure how to generate that iso, there's an example script under contrib, inside the kvm test directory. Here is an example of how the files inside this cd would be organized, assuming the iso image is mounted under `/tmp/virtio-win` (the actual cd has more files, but we took only the parts that concern to the example, win7 64 bits).

```
/tmp/virtio-win/
/tmp/virtio-win/vista
/tmp/virtio-win/vista/amd64
/tmp/virtio-win/vista/amd64/netkvm.cat
/tmp/virtio-win/vista/amd64/netkvm.inf
/tmp/virtio-win/vista/amd64/netkvm.pdb
/tmp/virtio-win/vista/amd64/netkvm.sys
/tmp/virtio-win/vista/amd64/netkvmco.dll
/tmp/virtio-win/vista/amd64/readme.doc
/tmp/virtio-win/win7
/tmp/virtio-win/win7/amd64
/tmp/virtio-win/win7/amd64/balloon.cat
/tmp/virtio-win/win7/amd64/balloon.inf
/tmp/virtio-win/win7/amd64/balloon.pdb
/tmp/virtio-win/win7/amd64/balloon.sys
/tmp/virtio-win/win7/amd64/blnsvr.exe
/tmp/virtio-win/win7/amd64/blnsvr.pdb
/tmp/virtio-win/win7/amd64/vioser.cat
/tmp/virtio-win/win7/amd64/vioser.inf
/tmp/virtio-win/win7/amd64/vioser.pdb
/tmp/virtio-win/win7/amd64/vioser.sys
/tmp/virtio-win/win7/amd64/vioser-test.exe
/tmp/virtio-win/win7/amd64/vioser-test.pdb
/tmp/virtio-win/win7/amd64/viostor.cat
/tmp/virtio-win/win7/amd64/viostor.inf
/tmp/virtio-win/win7/amd64/viostor.pdb
```

(continues on next page)

(continued from previous page)

```
/tmp/virtio-win/win7/amd64/viostor.sys
/tmp/virtio-win/win7/amd64/wdfcoinstaller01009.dll
...
```

If you are planning on installing WinXP or Win2003, you should also have a pre-made floppy disk image with the virtio drivers *and* a configuration file that the installer program will read to fetch the right drivers from it. Unfortunately, I don't have much info on how to build that file, you probably would have the image already assembled if you are willing to test those guest OS.

So you have to map the paths of your cd containing the drivers on the config variables. We hope to improve this in cooperation with the virtio drivers team.

6.11.2 Step by step procedure

We are assuming you already have the virtio cd properly assembled with you, as well as windows iso files that *do match the ones provided in our test_base.cfg.sample*. Don't worry though, we try as much as possible to use files from MSDN, to standardize.

We will use win7 64 bits (non sp1) as the example, so the CD you'd need is:

```
cdrom_cd1 = isos/windows/en_windows_7_ultimate_x86_dvd_x15-65921.iso
sha1sum_cd1 = 5395dc4b38f7bdb1e005ff414deedfdb16dbf610
```

This file can be downloaded from the MSDN site, so you can verify the SHA1 sum of it matches.

1. Git clone autotest to a convenient location, say $HOME/Code/autotest. See *the download source documentation* Please do use git and clone the repo to the location mentioned.
2. Execute the `get_started.py` script (see the *get started documentation <../GetStartedGuide>*. It will create the directories where we expect the cd files to be available. You don't need to download the Fedora 14 DVD, but you do need to download the winutils.iso. Our configuration is based on RHEL/Fedora based version that can be obtained from *here <https://docs.fedoraproject.org/en-US/quick-docs/creating-windows-virtual-machines-using-virtio-drivers/index.html>* and need to be placed in `/tmp/kvm_autotest_root/isos/windows`). Please, do read the documentation mentioned on the script to avoid missing packages installed and other misconfiguration.
3. Create a windows dir under `/tmp/kvm_autotest_root/isos`
4. Copy your windows 7 iso to `/tmp/kvm_autotest_root/isos/windows`
5. Edit the file cdkeys.cfg and put the windows 7 64 bit key on that file
6. Edit the file win-virtio.cfg and verify if the paths are correct. You can see that by looking this session:

```
64:
    unattended_install.cdrom, whql.support_vm_install:
        # Look at your cd structure and see where the drivers are
        # actually located (viostor and netkvm)
        virtio_storage_path = 'F:\win7\amd64'
        virtio_network_path = 'F:\vista\amd64'

        # Uncomment if you have a nw driver installer on the iso
        #virtio_network_installer_path = 'F:\RHEV-Network64.msi'
```

7. If you are using the cd with the layout mentioned on the beginning of this article, the paths are already correct. However, if they're different (more likely), you have to adjust paths. Don't forget to read and do all the config on win-virtio.cfg file as instructed by the comments.

8. On tests.cfg, you have to enable virtio install of windows 7. On the block below, you have to change `only rtl8139` to `only virtio_net` and `only ide` to `only virtio-blk`. You are informing autotest that you only want a vm with virtio hard disk and network device installed.

```
# Runs qemu-kvm, Windows Vista 64 bit guest OS, install, boot, shutdown
- @qemu_kvm_windows_quick:
    # We want qemu-kvm for this run
    qemu_binary = /usr/bin/qemu-kvm
    qemu_img_binary = /usr/bin/qemu-img
    # Only qcow2 file format
    only qcow2
    # Only rtl8139 for nw card (default on qemu-kvm)
    only rtl8139
    # Only ide hard drives
    only ide
    # qemu-kvm will start only with -smp 2 (2 processors)
    only smp2
    # No PCI assignable devices
    only no_pci_assignable
    # No large memory pages
    only smallpages
    # Operating system choice
    only Win7.64
    # Subtest choice. You can modify that line to add more subtests
    only unattended_install.cdrom, boot, shutdown
```

9. You have to change the bottom of tests.cfg to look like the below, Which means you are informing autotest to only run the test set mentioned above, rather than the default, that installs Fedora 15.

```
only qemu_kvm_windows_quick
```

10. As informed on the output of `get_started.py`, the command you can execute to run autotest is (please run this AS ROOT or sudo)

```
$ $HOME/Code/autotest/client/bin/autotest $HOME/Code/autotest/client/tests/kvm/
↪control
```

11. Profit! You automated install of Windows 7 with the virtio drivers will be carried out.

If you want to install other guests, as you might imagine, you can change `only Win7.64` with other guests, say `only Win2008.64.sp2`. Now, during the first time you perform your installs, it's good to watch the installation to see if there aren't problems such as a **wrong cd key** preventing your install from happening. Avocado-VT prints the qemu command line used, so you can see which vnc display you can connect to to watch your vm being installed.

6.12 Running QEMU kvm-unit-tests

Currently there are two ways to run kvm-unit-tests. Newer one with per test results method is *Run kvm-unit-tests in avocado* and the older one *Run kvm-unit-tests in avocado-vt* which requires manual modifications and only reports overall results.

6.12.1 Run kvm-unit-tests in avocado

There is a contrib script to run kvm-unit-test using *external-runner* Avocado feature. It optionally downloads the latest kvm-unit-test from Git, compiles it and runs the tests inside avocado.

The contrib script is located in `$AVOCADO/contrib/testsuites/run-kvm-unit-test.sh` or can be downloaded from: https://raw.githubusercontent.com/avocado-framework/avocado/master/contrib/testsuites/run-kvm-unit-test.sh

Then you simply run it and wait for results:

```
$ ./contrib/testsuites/run-kvm-unit-test.sh
JOB ID     : ca00d570f03b4942368ec9c407d69a881d98eb9d
JOB LOG    : /home/medic/avocado/job-results/job-2016-07-04T09.38-ca00d57/job.log
TESTS      : 38
 (01/38) access: PASS (4.77 s)
 (02/38) apic: PASS (5.10 s)
 (03/38) debug: PASS (1.82 s)
 (04/38) emulator:  ^C
Interrupt requested. Waiting 2 seconds for test to finish (ignoring new Ctrl+C until␣
↪then)

INTERRUPTED (0.84 s)
RESULTS    : PASS 3 | ERROR 0 | FAIL 0 | SKIP 34 | WARN 0 | INTERRUPT 1
JOB HTML   : /home/medic/avocado/job-results/job-2016-07-04T09.38-ca00d57/html/
↪results.html
TESTS TIME : 12.54 s
```

Note: You can specify various options including the avocado arguments. Use the *-h* to see them all (eg. wildcard to specify tests pattern or path to avoid (re)downloading of the kvm-unit-test)

6.12.2 Run kvm-unit-tests in avocado-vt

For a while now, qemu-kvm does contain a unittest suite that can be used to assess the behavior of some KVM subsystems. Ideally, they are supposed to PASS, provided you are running both the latest qemu and the latest linux KVM tree. Avocado-VT for quite a long time has support for running them in an automated way. It's a good opportunity to put your git branch to unittest, starting from a clean state (KVM autotest will fetch from your git tree, leaving your actual development tree intact and doing things from scratch, and that is less likely to mask problems).

A bit of context on Avocado-VT build tests

People usually don't know that Avocado-VT has support to build and install QEMU/KVM for testing purposes, from many different software sources. You can:

1. Build qemu-kvm from a git repo (most common choice for developers hacking on code)
2. Install qemu-kvm from an rpm file (people testing a newly built rpm package)
3. Install qemu-kvm straight from the Red Hat build systems (Koji is the instance of the build system for Fedora, Brew is the same, but for RHEL. With this we can perform quality control on both Fedora and RHEL packages, trying to anticipate breakages before the packages hit users)

For this article, we are going to focus on git based builds. Also, we are focusing on Fedora and RHEL. We'll try to write the article in a pretty generic fashion, you are welcome to improve this with details on how to do the same on your favorite linux distribution.

Before you start

You need to verify that you can actually build qemu-kvm from source, as well as the unittest suite.

1. Make sure you have the appropriate packages installed. You can read *the install prerequesite packages (client) section* for more information.

Step by step procedure

1. Git clone autotest to a convenient location, say `$HOME/Code/autotest`. See *the download source documentation* Please do use git and clone the repo to the location mentioned.
2. Execute the `get_started.py` script (see *the get started documentation*. If you just want to run unittests, you can safely skip each and every iso download possible, as *qemu-kvm will straight boot small kernel images (the unittests)* rather than full blown OS installs.
3. As running unittests is something that's fairly independent of other Avocado-VT testing you can do, and it's something people are interested in, we prepared a *special control file* and a *special configuration file* for it. On the kvm directory, you can see the files `unittests.cfg` `control.unittests`. You only need to edit `unittests.cfg`.
4. The file `unittests.cfg` is a stand alone configuration for running unittests. It is comprised by a build variant and a unittests variant. Edit the file, it'll look like:

```
... bunch of params needed for the Avocado-VT preprocessor
# Tests
variants:
    - build:
        type = build
        vms = ''
        start_vm = no
        # Load modules built/installed by the build test?
        load_modules = no
        # Save the results of this build on test.resultsdir?
        save_results = no
        variants:
            - git:
                mode = git
                user_git_repo = git://git.kernel.org/pub/scm/virt/kvm/qemu-kvm.git
                user_branch = next
                user_lbranch = next
                test_git_repo = git://git.kernel.org/pub/scm/virt/kvm/kvm-unit-
↪tests.git

    - unittest:
        type = unittest
        vms = ''
        start_vm = no
        unittest_timeout = 600
        testdev = yes
        extra_params += " -S"
        # In case you want to execute only a subset of the tests defined on the
        # unittests.cfg file on qemu-kvm, uncomment and edit test_list
        #test_list = idt_test hypercall vmexit realmode

only build.git unittest
```

5. As you can see above, you have places to specify both the userspace git repo and the unittest git repo. You are then free to replace `user_git_repo` with your own git repo. It can be a remote git location, or it can simply be the path to a cloned tree inside your development machine.
6. As of Fedora 15, that ships with gcc 4.6.0, the compilation is more strict, so things such as an unused variable in

the code *will* lead to a build failure. You can disable that level of strictness by providing *extra configure script options* to your qemu-kvm userspace build. Right below the `user_git_repo line`, you can set the variable `extra_configure_options` to include `--disable-werror`. Let's say you also want Avocado-VT to fetch from my local tree, `/home/lmr/Code/qemu-kvm`, master branch, same for the kvm-unit-tests repo. If you make those changes, your build variant will look like:

```
- git:
    mode = git
    user_git_repo = /home/lmr/Code/qemu-kvm
    extra_configure_options = --disable-werror
    user_branch = master
    user_lbranch = master
    test_git_repo = /home/lmr/Code/kvm-unit-tests
```

7. Now you can just run Avocado-VT as usual, you just have to change the main control file (called `control` with the unittest one `control.unittests`

```
$ $HOME/Code/autotest/client/bin/autotest $HOME/Code/autotest/client/tests/kvm/
↪control.unittests
```

8. The output of a typical unittest execution looks like. Notice that autotest informs you where the logs of each individual unittests are located, so you can check that out as well.

```
07/14 18:49:44 INFO |  unittest:0052| Running apic
07/14 18:49:44 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/apic.flat' -vnc :0 -chardev file,id=testlog,path=/
↪tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S -cpu␣
↪qemu64,+x2apic
07/14 18:49:46 INFO |  unittest:0096| Waiting for unittest apic to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 18:59:46 ERROR|  unittest:0108| Exception happened during apic: Timeout␣
↪elapsed (600s)
07/14 18:59:46 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/apic.log
07/14 18:59:46 INFO |  unittest:0052| Running smptest
07/14 19:00:15 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:00:16 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/smptest.flat' -vnc :0 -chardev file,id=testlog,
↪path=/tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:00:17 INFO |  unittest:0096| Waiting for unittest smptest to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:00:17 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:00:18 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/smptest.log
07/14 19:00:18 INFO |  unittest:0052| Running smptest3
07/14 19:00:18 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 3 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/smptest.flat' -vnc :0 -chardev file,id=testlog,
↪path=/tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
```

(continues on next page)

(continued from previous page)

```
07/14 19:00:19 INFO |  unittest:0096| Waiting for unittest smptest3 to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:00:19 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:00:20 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/smptest3.
↪log
07/14 19:00:20 INFO |  unittest:0052| Running vmexit
07/14 19:00:20 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/vmexit.flat' -vnc :0 -chardev file,id=testlog,
↪path=/tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:00:21 INFO |  unittest:0096| Waiting for unittest vmexit to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:00:31 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:00:31 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/vmexit.log
07/14 19:00:31 INFO |  unittest:0052| Running access
07/14 19:00:31 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/access.flat' -vnc :0 -chardev file,id=testlog,
↪path=/tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:00:32 INFO |  unittest:0096| Waiting for unittest access to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:01:02 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:01:03 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/access.log
07/14 19:01:03 INFO |  unittest:0052| Running emulator
07/14 19:01:03 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/emulator.flat' -vnc :0 -chardev file,id=testlog,
↪path=/tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:01:05 INFO |  unittest:0096| Waiting for unittest emulator to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:01:06 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:01:07 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/emulator.
↪log
07/14 19:01:07 INFO |  unittest:0052| Running hypercall
07/14 19:01:07 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/hypercall.flat' -vnc :0 -chardev file,id=testlog,
↪path=/tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:01:08 INFO |  unittest:0096| Waiting for unittest hypercall to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:01:08 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
```

(continues on next page)

(continued from previous page)

```
07/14 19:01:09 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/hypercall.
↪log
07/14 19:01:09 INFO |  unittest:0052| Running idt_test
07/14 19:01:09 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/idt_test.flat' -vnc :0 -chardev file,id=testlog,
↪path=/tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:01:10 INFO |  unittest:0096| Waiting for unittest idt_test to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:01:10 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:01:11 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/idt_test.
↪log
07/14 19:01:11 INFO |  unittest:0052| Running msr
07/14 19:01:11 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/msr.flat' -vnc :0 -chardev file,id=testlog,path=/
↪tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:01:12 INFO |  unittest:0096| Waiting for unittest msr to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:01:13 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:01:13 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/msr.log
07/14 19:01:13 INFO |  unittest:0052| Running port80
07/14 19:01:13 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/port80.flat' -vnc :0 -chardev file,id=testlog,
↪path=/tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:01:14 INFO |  unittest:0096| Waiting for unittest port80 to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:01:31 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:01:32 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/port80.log
07/14 19:01:32 INFO |  unittest:0052| Running realmode
07/14 19:01:32 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/realmode.flat' -vnc :0 -chardev file,id=testlog,
↪path=/tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:01:33 INFO |  unittest:0096| Waiting for unittest realmode to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:01:33 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:01:34 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/realmode.
↪log
```

(continues on next page)

(continued from previous page)

```
07/14 19:01:34 INFO |  unittest:0052| Running sieve
07/14 19:01:34 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/sieve.flat' -vnc :0 -chardev file,id=testlog,path=/
↪tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:01:35 INFO |  unittest:0096| Waiting for unittest sieve to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:02:05 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:02:05 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/sieve.log
07/14 19:02:05 INFO |  unittest:0052| Running tsc
07/14 19:02:05 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/tsc.flat' -vnc :0 -chardev file,id=testlog,path=/
↪tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:02:06 INFO |  unittest:0096| Waiting for unittest tsc to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:02:06 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:02:07 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/tsc.log
07/14 19:02:07 INFO |  unittest:0052| Running xsave
07/14 19:02:07 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/xsave.flat' -vnc :0 -chardev file,id=testlog,path=/
↪tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:02:08 INFO |  unittest:0096| Waiting for unittest xsave to complete,␣
↪timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:02:09 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:02:09 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/xsave.log
07/14 19:02:09 INFO |  unittest:0052| Running rmap_chain
07/14 19:02:09 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/rmap_chain.flat' -vnc :0 -chardev file,id=testlog,
↪path=/tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S
07/14 19:02:11 INFO |  unittest:0096| Waiting for unittest rmap_chain to complete,
↪ timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:02:12 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:02:13 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/rmap_chain.
↪log
07/14 19:02:13 INFO |  unittest:0052| Running svm
07/14 19:02:13 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/svm.flat' -vnc :0 -chardev file,id=testlog,path=/
↪tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S  enable
↪nesting -cpu qemu64,+svm
```

(continues on next page)

(continued from previous page)

```
07/14 19:02:13 INFO |   aexpect:0783| (qemu) qemu: -enable-nesting: invalid option
07/14 19:02:13 INFO |   aexpect:0783| (qemu) (Process terminated with status 1)
07/14 19:02:13 ERROR|  unittest:0108| Exception happened during svm: VM creation␣
↪command failed:    "/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -
↪vga std -monitor unix:'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,
↪nowait -qmp unix:'/tmp/monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -
↪serial unix:'/tmp/serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -
↪kernel '/usr/local/autotest/tests/kvm/unittests/svm.flat' -vnc :0 -chardev file,
↪id=testlog,path=/tmp/testlog-20110714-184944-6ms0 -device testdev,
↪chardev=testlog  -S -enable-nesting -cpu qemu64,+svm"    (status: 1,    output:
↪'qemu: -enable-nesting: invalid option\n')
07/14 19:02:13 ERROR|  unittest:0115| Not possible to collect logs
07/14 19:02:13 INFO |  unittest:0052| Running svm-disabled
07/14 19:02:13 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/svm.flat' -vnc :0 -chardev file,id=testlog,path=/
↪tmp/testlog-20110714-184944-6ms0 -device testdev,chardev=testlog  -S -cpu␣
↪qemu64,-svm
07/14 19:02:14 INFO |  unittest:0096| Waiting for unittest svm-disabled to␣
↪complete, timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:02:15 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:02:16 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/svm-
↪disabled.log
07/14 19:02:16 INFO |  unittest:0052| Running kvmclock_test
07/14 19:02:16 INFO |    kvm_vm:0782| Running qemu command:
/usr/local/autotest/tests/kvm/qemu -name 'vm1' -nodefaults -vga std -monitor unix:
↪'/tmp/monitor-humanmonitor1-20110714-184944-6ms0',server,nowait -qmp unix:'/tmp/
↪monitor-qmpmonitor1-20110714-184944-6ms0',server,nowait -serial unix:'/tmp/
↪serial-20110714-184944-6ms0',server,nowait -m 512 -smp 2 -kernel '/usr/local/
↪autotest/tests/kvm/unittests/kvmclock_test.flat' -vnc :0 -chardev file,
↪id=testlog,path=/tmp/testlog-20110714-184944-6ms0 -device testdev,
↪chardev=testlog  -S --append "10000000 `date +%s`"
07/14 19:02:17 INFO |  unittest:0096| Waiting for unittest kvmclock_test to␣
↪complete, timeout 600, output in /tmp/testlog-20110714-184944-6ms0
07/14 19:02:33 INFO |   aexpect:0783| (qemu) (Process terminated with status 0)
07/14 19:02:34 INFO |  unittest:0113| Unit test log collected and available under␣
↪/usr/local/autotest/results/default/kvm.qemu-kvm-git.unittests/debug/kvmclock_
↪test.log
07/14 19:02:34 ERROR|       kvm:0094| Test failed: TestFail: Unit tests failed:␣
↪apic svm
```

You might take a look at the `unittests.cfg` config file options to do some tweaking you might like, such as making the timeout to consider a unittest as failed smaller and other things.

6.13 Parallel Jobs

Avocado-VT ships with a plugin that creates a lock file in a known public location (`/tmp` by default, but configurable) to prevent multiple runs of jobs that include VT tests.

The reason is that, by default, multiple jobs running at the same can access the same data files and cause corruption. Example of data files are the guest images, which are usually modified, either directly or indirectly by the tests.

6.13.1 Checking Installation

The vt-joblock is installed and registered by default. To make sure it's active, run:

```
$ avocado plugins
```

The VT Job lock plugin should be listed:

```
Plugins that run before/after the execution of jobs (avocado.plugins.job.prepost):
...
vt-joblock Avocado-VT Job Lock/Unlock
...
```

6.13.2 Configuration

The configuration for the vt-joblock plugin can be found at `/etc/avocado/conf.d/vt_joblock.conf`. Example of a configuration file content follows:

```
[plugins.vtjoblock]
# Directory where the lock file will be located. Avocado should have permission
# to write to this directory.
dir=/tmp
```

The configuration key `dir` lets you set the directory where Avocado will look for an existing lock file before running, and create one if it doesn't exist yet.

6.13.3 Running Parallel Jobs

Supposing that you have multiple users on a single machine, using different data directories, you can allow parallel VT jobs by setting different lock directories for each user.

To do so, you can add the customized lock directory to the user's own Avocado configuration file. Start by creating a lock directory:

```
[user1@localhost] $ mkdir ~/avocado/data/avocado-vt/lockdir
```

Then modify the user's own configuration to point to the newly created lock directory:

```
[user1@localhost] $ cat >> ~/.config/avocado/avocado.conf <<EOF
[plugins.vtjoblock]
dir=/home/user1/avocado/data/avocado-vt/lockdir
EOF
```

Then verify with:

```
[user1@localhost] $ avocado config | grep plugins.vtjoblock
...
plugins.vtjoblock.dir          /home/user1/avocado/data/avocado-vt/lockdir
...
```

Do the same thing for other users and their jobs will not be locked by one another.

6.14 Running in emulation mode (TCG)

Sometimes it's useful to run Avocado-vt in emulation mode (eg. when checking PR about different architecture, or to debug certain feature while executing the test). As this is not default, extra arguments are usually required.

Shared for all architectures is that you need the image. We supply JeOS for most of the architectures, so you should be able to get it via:

```
$ avocado vt-bootstrap --vt-guest-os JeOS.$VERSION.$ARCH
```

where:

- `VERSION` is JeOS version (when writing this document it was `27`)
- `ARCH` is the desired architecture (eg. aarch64)

Or you can run any of the `unattended_install` tests with `--vt-guest-os` of your choice (very slow).

When running the tests, on top of the usual arguments, you usually need to specify:

- `--vt-qemu-bin` - path to binary that supports expected architecture (eg. `/usr/local/bin/qemu-system-aarch64`)
- `--vt-qemu-dst-bin` - for migration tests you need also to specify the destination qemu binary (otherwise default one is used)
- `enable_kvm=no` - to disable KVM (if necessary)

Note: Some test require additional dependencies and are marked as `no JeOS` (another group is marked as `only RHEL`) but it might be useful for debugging purposes to use them with JeOS. You can do that by symlinking the `$avocado-vt-data/images/jeos-$version-$arch.qcow2` to `$avocado-vt-data/images/rhel${version}devel-$arch.qcow2` and using the boot test with `--vt-guest-os RHEL.$version`. To add extra packages use `ctrl+Z` when it's about to ssh in. Then you can ssh to that guest from your machine, run `dnf install ...` to install the extra packages (`gcc` suffices for most test), shut the machine down, backup it to `$name.backup` and resume the `boot` test by `ctrl+Z`. Obviously the test will fail, refreshes the image from `$name.backup` but since then you have slightly fattier JeOS symlinked to RHEL capable of running some extended tests without the need to run full installation in TCG mode. Beware, vt-bootstrap might overwrite the *.backup* from archive.

6.14.1 aarch64

ARM always requires *cpu_model* as well as *machine_type*. To get list of available models you can run `qemu-system-aarch64 -cpu help -M virt` (note: not all listed cpus are bootable). By default Avocado-vt uses `-machine $machine_type,gic-version=host` to use host's GIC version, but this is not possible to evaluate in TCG (especially without GIC on x86) so one needs to either pick a fixed version or simply use qemu default by cleaning the `machine_type_extra_params`. Complete example would be:

```
$ avocado vt-bootstrap --vt-guest-os JeOS.27.aarch64
$ avocado --show all run --vt-extra-params enable_kvm=no cpu_model=cortex-a57 \
    machine_type_extra_params='' --vt-machine-type aarch64 --vt-arch arm64-pci \
    --vt-qemu-bin /usr/local/bin/qemu-system-aarch64 -- boot
```

6.14.2 ppc64/ppc64le

PowerPC can use either BE or LE instructions, but from qemu point of view nothing changes. Still for Avocado-vt you either have to specify `--vt-arch ppc64` or `--vt-arch ppc64le` to choose the right distribution image (both were available as JeOS when writing this document). Apart from this no additional tweaks are necessary:

```
$ avocado vt-bootstrap --vt-guest-os JeOS.27.ppc64
$ avocado --show all run --vt-extra-params enable_kvm=no --vt-arch ppc64 \
    --vt-machine-type pseries --vt-qemu-bin /usr/local/bin/qemu-system-ppc64 -- boot

$ avocado vt-bootstrap --vt-guest-os JeOS.27.ppc64le
$ avocado --show all run --vt-extra-params enable_kvm=no --vt-arch ppc64le \
    --vt-machine-type pseries --vt-qemu-bin /usr/local/bin/qemu-system-ppc64 -- boot
```

6.14.3 s390x

For KVM execution Avocado-vt uses `-cpu host` on s390x, which is not possible without KVM. To execute in TCG mode you need to replace it with either a supported CPU type or simply leave it blank:

```
$ avocado vt-bootstrap --vt-guest-os JeOS.27.s390x
$ avocado --show all run --vt-extra-params enable_kvm=no cpu_model='' --vt-arch s390x␣
↪\
    --vt-machine-type s390-virtio --vt-qemu-bin /usr/local/bin/qemu-system-s390x --␣
↪boot
```

6.14.4 riscv64

When writing this document, riscv64 was not available as JeOS and even Fedora support was not straight forward. See **'riscv64_setup'_** for setup instructions. Apart from the setup running riscv64 does not require any additional arguments:

```
$ avocado run --vt-machine-type riscv64-mmio --vt-arch riscv64 \
    --vt-extra-params enable_kvm=no --vt-guest-os Fedora.28 -- boot
```

6.15 Contribution and Community Guide

Contents:

6.15.1 Contact information

Currently there is no dedicated Avocado-vt communication channel, but you can use main Avocado-framework channels to reach us:

- Avocado mailing list (hosted by red hat)
- IRC channel: irc.oftc.net #avocado

6.15.2 Downloading the Source

The main source is maintained on git and may be cloned as below:

```
git clone https://github.com/avocado-framework/avocado-vt.git
```

If you want to learn how to use git as an effective contribution tool, consider reading the git workflow autotest docs

6.15.3 Avocado VT Development Workflow

http://avocado-framework.readthedocs.org/en/latest/ContributionGuide.html

6.15.4 Contributions Guidelines and Tips

Code

Contributions of additional tests and code are always welcome. If in doubt, and/or for advice on approaching a particular problem, please contact the projects members (see section _collaboration) Before submitting code, please review the git repository configuration guidelines.

To submit changes, please follow these instructions. Please allow up to two weeks for a maintainer to pick up and review your changes. Though, if you'd like help at any stage, feel free to post on the mailing lists and reference your pull request.

Docs

Please edit the documentation directly to correct any minor inaccuracies or to clarify items. The preferred markup syntax is ReStructuredText, keeping with the conventions and style found in existing documentation. For any graphics or diagrams, web-friendly formats should be used, such as PNG or SVG.

Avoid using 'you', 'we', 'they', as they can be ambiguous in reference documentation. It works fine in conversation and e-mail, but looks weird in reference material. Similarly, avoid using 'unnecessary', off-topic, or extra language. For example in American English, "Rinse and repeat" is a funny phrase, but could cause problems when translated into other languages. Basically, try to avoid anything that slows the reader down from finding facts.

Rules for Reviewers

1. Everyone who has experiences in the project is encouraged to review PRs.
2. Respectful, kind, patient to the coders
3. Freely deny for changes the codebase does not want/need even though perfect design/codes
4. Ask questions rather than make statements.
5. Not encourage the "Why" questions. Good practice: e.g. Wouldn't it make more sense to Would you like to? Could you give the reason that . . . ?
6. Remember to praise.
7. Remember that there is often more than one way to approach a solution.
8. Given clear and useful comments, and explain the reason why reqest change

9. In general, reviewers should favor approving a PR once it is in a state where it definitely improves the overall code health of the system being worked on, even if the PR isn't perfect (maintainability, readability, and understandability).
10. Share your best practice/knowledge as a mentor.
11. Cautiously regard personal preference as best practice and impose to contributors
12. Dismiss your approval if you add new comment for a PR after you already have given an approval
13. It is ok to use a 'request review' tool to ask someone to review as you like, but not a must.
14. It is ok to cancel the request review to you if you think you are not a suitable one for this PR.
15. Wait for request review for no more than 2 weeks on those PRs which have already 2 approvals

Rules for Maintainers

1. Includes all reviewer's rules
2. Make sure all PRs submitted can be closed within 3 months
3. Generally every PR needs at least 1 maintainer's approval and total 2 approvals before being merged.

a) Add request for review for more maintainers if there is a need

b) Add a comment to explain why a PR need the label 'request_2_maintainers'

4. Mark proper and necessary labels according to the information provided by the PR contributor
5. Closing a PR threshold:no response from contributor after 1 month, the PR will be labelled as, "No response" and after 3 months it will be closed

Rules for Contributors

1. [Must] Coding style compliance, for example, align with inspekt tool, pep8
2. [Must] PR commit message is meaningful. Refer to the link on how to write a good commit message
3. [Must] Travis CI pass and no conflict
4. [Must] Provide test results. If no, provide justification. Apply to any PR

   One of below options should be aligned:

5. [Must] If the function defined with right docstring (description and params, and return if have)
6. [Must] If the PR depends on other PRs, please add a comment to say if your PR has a dependence in order to ensure the PR is merged after dependence PRs
7. [Must] If the API of one library is changed, ask for all test cases to be modified which invoke this library and provide test results of representative test cases.
8. [Must] If the case does some package version judgement for the new case support or compatible backwards
9. [Must] If the test code have suitable env backup and recovery steps
10. [Optional] If have the necessary and clear comments for the code explanation for steps
11. [Optional] If the case is applied to multiple arches.
12. [Optional] If have duplication, need to create new function or reuse existing library in avocado/avocado-vt
13. [Optional] If the logic are complete and no important branches which are not dealt with
14. [Optional] If the code seems clear and concise, define functions to increase the readability

15. [Optional] Use python supported library instead of shell cmd running by process.run if possible
16. [Optional] If the feature test related aspects are correct
17. [Optional] Add comments to ask questions which you do not understand
18. [Optional] Pay more attention to 'test.fail(xxx)' or exception raise part, such as if there is log info
19. [Optional] Reply to the comment when you have fixed the comment (see good sample in Appendix.4)
20. [Optional] Better to use @Someone to ask for review when your PR is submitted
21. [Optional] Use 'request review' to ask the original reviewer to request again when you finish updates

6.16 Experimental features

6.16.1 riscv64

The support for riscv64 is very experimental and requires special preparations. Basically you need to prepare your system according to:

https://fedorapeople.org/groups/risc-v/disk-images/readme.txt

Which means you need to install latest qemu-system-riscv (tested with qemu `00928a421d47f49691cace1207481b7aad31b1f1`) or install the one provided by Rich:

https://copr.fedorainfracloud.org/coprs/rjones/riscv/

And you need to download a suitable image and bootable kernel to the right location:

- kernel: https://fedorapeople.org/groups/risc-v/disk-images/bbl needs to be downloaded in `$AVOCADO_VT_DATA/images/f28-riscv64-kernel`
- image: https://fedorapeople.org/groups/risc-v/disk-images/stage4-disk.img.xz needs to be downloaded in `$AVOCADO_VT_DATA/images/`, extracted and converted to `qcow2` using name `f28-riscv64.qcow2`.

Basically you can go into `$AVOCADO_VT_DATA/images` and execute:

```
curl https://fedorapeople.org/groups/risc-v/disk-images/bbl -o f28-riscv64-kernel
curl https://fedorapeople.org/groups/risc-v/disk-images/stage4-disk.img.xz | xz -d >␣
↪stage4-disk.img
qemu-img convert -f raw -O qcow2 stage4-disk.img f28-riscv64.qcow2
rm stage4-disk.img
```

Also I'd recommend booting the guest:

```
qemu-system-riscv64 \
    -nographic \
    -machine virt \
    -smp 4 \
    -m 2G \
    -kernel f28-riscv64-kernel \
    -object rng-random,filename=/dev/urandom,id=rng0 \
    -device virtio-rng-device,rng=rng0 \
    -append "console=ttyS0 ro root=/dev/vda" \
    -device virtio-blk-device,drive=hd0 \
    -drive file=f28-riscv64.qcow2,format=qcow2,id=hd0 \
    -device virtio-net-device,netdev=usernet \
    -netdev user,id=usernet,hostfwd=tcp::10000-:22
```

and running the Fedora-25.ks post-install steps:

```
dnf -y install @standard @c-development @development-tools python net-tools sg3_utils␣
↪python-pip
grubby --remove-args="rhgb quiet" --update-kernel=$(grubby --default-kernel)
dhclient
chkconfig sshd on
iptables -F
systemctl mask tmp.mount
echo 0 > /selinux/enforce
sed -i "/^HWADDR/d" /etc/sysconfig/network-scripts/ifcfg-eth0
# if package groups were missing from main installation repo
# try again from installed system
dnf -y groupinstall c-development development-tools
# include avocado: allows using this machine with remote runner
# Fallback to pip as it's not yet built for riscv64
dnf -y install python2-avocado || pip install python2-avocado
```

Tip: If you want to use riscv without kvm (eg. on x86 host) use something like `avocado run --vt-machine-type riscv64-mmio --vt-arch riscv64 --vt-extra-params enable_kvm=no --vt-guest-os Fedora.28 -- boot` which sets the right machine/arch and disables kvm (uses tcg).

CHAPTER 7

API Reference

7.1 virttest

7.1.1 virttest package

Subpackages

virttest.libvirt_xml package

Subpackages

virttest.libvirt_xml.devices package

Submodules

virttest.libvirt_xml.devices.address module

virttest.libvirt_xml.devices.audio module

virttest.libvirt_xml.devices.base module

virttest.libvirt_xml.devices.channel module

virttest.libvirt_xml.devices.character module

virttest.libvirt_xml.devices.console module

virttest.libvirt_xml.nwfilter_binding module

virttest.libvirt_xml.nwfilter_xml module

virttest.libvirt_xml.pool_xml module

virttest.libvirt_xml.secret_xml module

virttest.libvirt_xml.snapshot_xml module

virttest.libvirt_xml.sysinfo_xml module

virttest.libvirt_xml.vm_xml module

virttest.libvirt_xml.vol_xml module

virttest.libvirt_xml.xcepts module

Module contents

virttest.qemu_devices package

Submodules

virttest.qemu_devices.qcontainer module

virttest.qemu_devices.qdevices module

virttest.qemu_devices.utils module

Shared classes and functions (exceptions, . . .)

copyright 2013 Red Hat Inc.

exception virttest.qemu_devices.utils.**DeviceError**
Bases: exceptions.Exception

General device exception

exception virttest.qemu_devices.utils.**DeviceHotplugError**(*device*, *reason*, *vmdev*, *ver_out=None*)
Bases: *virttest.qemu_devices.utils.DeviceInsertError*

Fail to hotplug device

exception virttest.qemu_devices.utils.**DeviceInsertError**(*device*, *reason*, *vmdev*)
Bases: *virttest.qemu_devices.utils.DeviceError*

Fail to insert device

exception virttest.qemu_devices.utils.**DeviceRemoveError**(*device*, *reason*, *vmdev*)
Bases: *virttest.qemu_devices.utils.DeviceInsertError*

Fail to remove device

exception virttest.qemu_devices.utils.**DeviceUnplugError**(*device*, *reason*, *vmdev*)
Bases: *virttest.qemu_devices.utils.DeviceHotplugError*

Fail to unplug device

virttest.qemu_devices.utils.**none_or_int**(*value*)
Helper function which returns None or int()

virttest.qemu_devices.utils.**set_cmdline_format_by_cfg**(*dev*, *config*, *key*)
Mark the cmdline format based on the settings.

Parameters

- **dev** – The device needed to be marked
- **config** – Dict with setting
- **key** – The field of device

Module contents

virttest.remote_commander package

Submodules

virttest.remote_commander.messenger module

Created on Dec 6, 2013

author jzupka, astepano

contact Andrei Stepanov <astepano@redhat.com>

class virttest.remote_commander.messenger.**DataWrapper**
Bases: object

Basic implementation of IOWrapper for stdio.

decode(*data*)
Decodes the data which was read.

Returns decoded data.

encode(*data*)
Encode data.

Returns encoded data.

class virttest.remote_commander.messenger.**DataWrapperBase64**
Bases: *virttest.remote_commander.messenger.DataWrapper*

Basic implementation of IOWrapper for stdio.

decode(*data*)
Decodes the data which was read.

Returns decoded data.

encode(*data*)
Encode data.

Returns encoded data.

class virttest.remote_commander.messenger.**IOWrapper**(*obj*)
Bases: object

Class encaptulates io operation to be more consist in different implementations. (stdio, sockets, etc..)

Parameters **obj** – IO obj for example file decriptor.

close()

fileno()
Function should return file descriptor number. If object should be used for standard io operation.

Returns File number.

read(*max_len*, *timeout=None*)
Read function should be reinmplemented as blocking reading from data source when timeout is None and nonblocking for timeout is not None. Implementation example StdIWrapper.

Params max_len Max len of readed data.

Parameters **timeout** (*float*) – Timeout of reading operation.

Returns Readed data.

write(*data*)
Write function should be implemented for object uded for writing.

Parameters **data** (*str.*) – Data to write.

class virttest.remote_commander.messenger.**Messenger**(*stdin*, *stdout*)
Bases: object

Class could be used for communication between two python process connected by communication canal wrapped by IOWrapper class. Pickling is used for communication and thus it is possible to communicate every picleable object.

Params stdin Object for read data from communication interface.

Params stdout Object for write data to communication interface.

close()

flush_stdin()
Flush all input data from communication interface.

format_msg(*data*)
Format message where first 10 char is length of message and rest is piclked message.

read_msg(*timeout=None*)
Read data from com interface.

Parameters **timeout** (*float*) – timeout for reading data.

Returns (True, data) when reading is successful. (False, None) when other side is closed. (None, None) when reading is timeouted.

write_msg(*data*)
Write formatted message to communication interface.

exception virttest.remote_commander.messenger.**MessengerError**(*msg*)
Bases: exceptions.Exception

class virttest.remote_commander.messenger.**StdIOWrapper**(*obj*)
Bases: *virttest.remote_commander.messenger.IOWrapper*, *virttest.remote_commander.messenger.DataWrapper*

Basic implementation of IOWrapper for stdio.

Parameters **obj** – IO obj for example file decriptor.

close()

fileno()
Function should return file descriptor number. If object should be used for standard io operation.

Returns File number.

class virttest.remote_commander.messenger.**StdIOWrapperIn**(*obj*)
Bases: *virttest.remote_commander.messenger.StdIOWrapper*

Basic implementation of IOWrapper for stdin

Parameters **obj** – IO obj for example file decriptor.

read(*max_len*, *timeout=None*)
Read function should be reinmplemented as blocking reading from data source when timeout is None and nonblocking for timeout is not None. Implementation example StdIWrapper.

Params max_len Max len of readed data.

Parameters **timeout** (*float*) – Timeout of reading operation.

Returns Readed data.

class virttest.remote_commander.messenger.**StdIOWrapperInBase64**(*obj*)
Bases: *virttest.remote_commander.messenger.StdIOWrapperIn*, *virttest.remote_commander.messenger.DataWrapperBase64*

Basic implementation of IOWrapper for stdin

Parameters **obj** – IO obj for example file decriptor.

class virttest.remote_commander.messenger.**StdIOWrapperOut**(*obj*)
Bases: *virttest.remote_commander.messenger.StdIOWrapper*

Basic implementation of IOWrapper for stdout

Parameters **obj** – IO obj for example file decriptor.

write(*data*)
Write function should be implemented for object uded for writing.

Parameters **data** (*str.*) – Data to write.

class virttest.remote_commander.messenger.**StdIOWrapperOutBase64**(*obj*)
Bases: *virttest.remote_commander.messenger.StdIOWrapperOut*, *virttest.remote_commander.messenger.DataWrapperBase64*

Basic implementation of IOWrapper for stdout

Parameters **obj** – IO obj for example file decriptor.

virttest.remote_commander.remote_interface module

Created on Dec 11, 2013

author jzupka, astepano

contact Andrei Stepanov <astepano@redhat.com>

class virttest.remote_commander.remote_interface.**BaseCmd**(*func_cmd*, **args*, ***kargs*)

Bases: *virttest.remote_commander.remote_interface.CmdMessage*

Class used for moving information about commands between master and slave.

args

cmd_hash

func

is_async()

Returns True if command is async else False

is_finished()

Returns True if command is finished else False

kargs

nh_stderr

nh_stdin

nh_stdout

results

single_cmd_id = 0

update(*basecmd*)

Sync local class with class moved over the messanger.

Parameters **basecmd** (BaseCmd) – basecmd from which should be sync data to this instance

update_cmd_hash(*basecmd*)

class virttest.remote_commander.remote_interface.**CmdMessage**(*cmd_id*)

Bases: object

Base cmd message class

cmd_id

isCmdMsg()

class virttest.remote_commander.remote_interface.**CmdQuery**(**args*, ***kargs*)

Bases: object

Command-msg-request from VM to avocado-vt test.

Command for asking from VM to avocado-vt.

Parameters

- **args** – Something pickable. Is irrelevant for messenger.
- **kargs** – Something pickable. Is irrelevant for messenger.

class virttest.remote_commander.remote_interface.**CmdRespond**(*respond*)

Bases: object

Command-msg-answer from avocado-test to VM.

Command for answering avocado-vt to VM.

Parameters **respond** – Something pickable. Is irrelevant for messenger.

exception virttest.remote_commander.remote_interface.**CmdTraceBack**(*msg*)
Bases: exceptions.Exception

Represent back-trace used for error tracing on remote side.

exception virttest.remote_commander.remote_interface.**CommanderError**(*msg*)
Bases: *virttest.remote_commander.remote_interface.MessengerError*

Represent error in Commander

exception virttest.remote_commander.remote_interface.**MessengerError**(*msg*)
Bases: exceptions.Exception

Represented error in messanger.

class virttest.remote_commander.remote_interface.**StdErr**(*msg*, *cmd_id=None*)
Bases: *virttest.remote_commander.remote_interface.StdStream*

Represent message from stderr string data from remote client

cmd_id

msg

class virttest.remote_commander.remote_interface.**StdOut**(*msg*, *cmd_id=None*)
Bases: *virttest.remote_commander.remote_interface.StdStream*

Represent message from stdout string data from remote client

cmd_id

msg

class virttest.remote_commander.remote_interface.**StdStream**(*msg*, *cmd_id=None*)
Bases: *virttest.remote_commander.remote_interface.CmdMessage*

Represent message string data from remote client

msg

virttest.remote_commander.remote_master module

Created on Dec 6, 2013

author jzupka, astepano

contact Andrei Stepanov <astepano@redhat.com>

class virttest.remote_commander.remote_master.**CmdEncapsulation**(*master*, *obj_name*, *name*)
Bases: object

Class parse command name cmd.nohup.shell -> [“nohup”, “shell”]

class virttest.remote_commander.remote_master.**CmdMaster**(*commander*, *name*, **args*, ***kargs*)
Bases: object

Representation of BaseCmd on master side.

Params commander Commander from which was command started.

Params name Name parsed to string representation

Parmas args list to arguments to cmd.

Params kargs {}

basecmd
Property basecmd getter

getbasecmd()
Property basecmd getter

getstderr()
Property stderr getter

getstdout()
Property stdout getter

send_stdin(*msg*)
Send data to stdin

set_commander(*commander*)
For nohup commands it allows connect cmd to new created commander.

setbasecmd(*value*)
Property basecmd setter _resuls_cnt identify if value was change from last reading.

setstderr(*value*)
Property stderr setter _stderr_cnt identify if value was change from last reading.

setstdout(*value*)
Property stdout setter _stdout_cnt identify if value was change from last reading.

stderr
Property stderr getter

stdout
Property stdout getter

wait()
Wait until command return results.

wait_response()
Wait until command return any cmd.

exception virttest.remote_commander.remote_master.**CmdTimeout**(*msg*)
Bases: *virttest.remote_commander.remote_interface.MessengerError*

Raised when waiting for cmd exceeds time define by timeout.

class virttest.remote_commander.remote_master.**Commander**
Bases: object

Commander representation for transfer over network.

class virttest.remote_commander.remote_master.**CommanderMaster**(*stdin*, *stdout*, *debug=False*)
Bases: *virttest.remote_commander.messenger.Messenger*

Class commander master is responsible for communication with commander slave. It invoke commands to slave part and receive messages from them. For communication is used only stdin and stdout which are streams from slave part.

close()

cmd(*cmd*)
Invoke command on client side.

listen_cmds(*cmd*)
Manage basecmds from slave side.

listen_errors(*cmd*)
Listen for errors raised from slave part of commander.

listen_messenger(*timeout=60*)
Wait for msg from slave side and take care about them.

listen_queries(*cmd*)
Manage queries from slave side.

listen_streams(*cmd*)
Listen on all streams included in Commander commands.

set_responder(*responder*)

Warning Users' helper becomes a part of a remote_commander. remote_commander is treated as a session. All sessions are part of BaseVM object. If you provide unpickable helper then your BaseVM also becomes unpickable. You can catch something like:

```
File "/usr/lib64/python2.7/copy_reg.py", line 74, in _reduce_ex
getstate = self.__getstate__
KeyError: '__getstate__'
```

For more info see:

```
https://github.com/avocado-framework/avocado-vt/issues/455
```

wait(*cmd*)
Wait until command return results.

wait_response(*cmd*)
Wait until command return any cmd.

virttest.remote_commander.remote_master.**getsource**(*obj*)

virttest.remote_commander.remote_master.**wait_timeout**(*timeout*)

virttest.remote_commander.remote_runner module

Created on Dec 6, 2013

author jzupka, astepano

contact Andrei Stepanov <astepano@redhat.com>

class virttest.remote_commander.remote_runner.**CmdFinish**(*parent=False*)
Bases: object

Class used for communication with child process. This class

pid

class virttest.remote_commander.remote_runner.**CmdSlave**(*baseCmd*)
Bases: object

Representation of BaseCmd on slave side.

Parameters **baseCmd** – basecmd for encapsulation.

close_pipes()
Close command communication pipe.

finish(*commander*)
Remove cmd from commander commands on finish of process.

parse_func_name(*func_name*, *commander*)
Parse name sended from master.

format: `["manage|async|nohup| ", "fnname1", "fnname2", ...]`

Parameters

- **func_name** – Function name
- **commander** – Where to execute the command (remote or local)

recover_fds()
Helper function for reconnect to daemon/nohup process.

recover_paths()
Helper function for reconnect to daemon/nohup process.

work()
Wait for message from running child process

class virttest.remote_commander.remote_runner.**CommanderSlave**(*stdin*, *stdout*, *o_stdout*, *o_stderr*)
Bases: *virttest.remote_commander.messenger.Messenger*

Class commander slace is responsible for communication with commander master. It invoke commands to slave part and receive messages from them. For communication is used only stdin and stdout which are streams from slave part.

cmd_loop()
Wait for commands from master and receive results and outputs from commands.

class virttest.remote_commander.remote_runner.**CommanderSlaveCmds**(*stdin*, *stdout*, *o_stdout*, *o_stderr*)
Bases: *virttest.remote_commander.remote_runner.CommanderSlave*

Class extends CommanderSlave and adds to them special commands like shell process, interactive python, send_msg to cmd.

add_function(*f_code*)
Adds function to client code.

Parameters **f_code** (*str.*) – Code of function.

copy_file(*name*, *path*, *content*)
Really naive implementation of copping files. Should be used only for short files.

exit()
Method for killing command slave.

import_src(*name*, *path=None*)
Import file to running python session.

interactive()
Starts interactive python.

python_file_run_with_helper(*test_path*)
Call run() function at external python module.

Parameters **test_path** – Path to python file. Call run() at this module.

Returns module.run().return

register_cmd(*basecmd*, *basecmd_cmd_id*)
Second side of set_commander cmd from master. It register existing cmd to CommandSlave dict.

Parameters

- **basecmd** (BaseCmd) – cmd which should be added to CommandSlave dict
- **basecmd_cmd_id** (*int*) – number under which should be stored

send_msg(*msg*, *cmd_id*)
Send msg to cmd with id == cmd_id

Parameters

- **msg** (*str*) – message passed to cmd over the stdin
- **cmd_id** – id of cmd.

shell(*cmd*)
Starts shell process. Stdout is automatically copied to basecmd.stdout

Parameters **cmd** – Command which should be started.

Returns basecmd with return code of cmd.

class virttest.remote_commander.remote_runner.**Helper**(*messenger*)
Bases: object

Passed to external test on VM.

flush_buf()

info(**args*, ***kargs*)

messenger

query_master(**args*, ***kargs*)
Read CmdRespond from master.

virttest.remote_commander.remote_runner.**clean_tmp_dir**(*path*)
Clean up directory.

virttest.remote_commander.remote_runner.**close_unused_fds**(*fds*)
Close all file descriptors which are not necessary anymore.

Parameters **fds** (*builtin.list*) – file descriptors

virttest.remote_commander.remote_runner.**create_process_cmd**()
Create child process without clean process data thanks that it is possible call function and classes from child process.

virttest.remote_commander.remote_runner.**daemonize**(*pipe_root_path='/tmp'*)
Init daemon.

Parameters **pipe_root_path** – path to directory for pipe.

Returns [True if child, stdin_path, stdou_path, stderr_path]

virttest.remote_commander.remote_runner.**gen_tmp_dir**(*root_path*)
Try to create tmp dir with special name.

virttest.remote_commander.remote_runner.**remote_agent**(*in_stream_cls*, *out_stream_cls*)
Connect file descriptors to right pipe and start slave command loop. When something happened it raise exception which could be caught by cmd master.

Params in_stream_cls Class encapsulated input stream.

Params out_stream_cls Class encapsulated output stream.

virttest.remote_commander.remote_runner.**sort_fds_event**(*fds*)

Module contents

virttest.staging package

Submodules

virttest.staging.lv_utils module

virttest.staging.service module

virttest.staging.utils_cgroup module

virttest.staging.utils_koji module

virttest.staging.utils_memory module

Module contents

virttest.test_setup package

Submodules

virttest.test_setup.core module

virttest.test_setup.networking module

virttest.test_setup.os_posix module

Module contents

virttest.tests package

Submodules

virttest.tests.unattended_install module

Module contents

virttest.unittest_utils package

Submodules

virttest.unittest_utils.mock module

exception virttest.unittest_utils.mock.**CheckPlaybackError**
Bases: exceptions.Exception

Raised when mock playback does not match recorded calls.

class virttest.unittest_utils.mock.**SaveDataAfterCloseStringIO**(*buf=''*)
Bases: StringIO.StringIO

Saves the contents in a final_data property when close() is called.

Useful as a mock output file object to test both that the file was closed and what was written.

Properties:

final_data: Set to the StringIO's getvalue() data when close() is called. None if close() has not been called.

close()

final_data = None

exception virttest.unittest_utils.mock.**StubNotFoundError**
Bases: exceptions.Exception

Raised when god is asked to unstub an attribute that was not stubbed

class virttest.unittest_utils.mock.**anything_comparator**
Bases: *virttest.unittest_utils.mock.argument_comparator*

is_satisfied_by(*parameter*)

class virttest.unittest_utils.mock.**argument_comparator**
Bases: object

is_satisfied_by(*parameter*)

class virttest.unittest_utils.mock.**base_mapping**(*symbol*, *return_obj*, **args*, ***dargs*)
Bases: object

match(**args*, ***dargs*)

class virttest.unittest_utils.mock.**equality_comparator**(*value*)
Bases: *virttest.unittest_utils.mock.argument_comparator*

is_satisfied_by(*parameter*)

class virttest.unittest_utils.mock.**function_any_args_mapping**(*symbol*, *return_val*, **args*, ***dargs*)
Bases: *virttest.unittest_utils.mock.function_mapping*

A mock function mapping that doesn't verify its arguments.

match(**args*, ***dargs*)

class virttest.unittest_utils.mock.**function_mapping**(*symbol*, *return_val*, **args*, ***dargs*)
Bases: *virttest.unittest_utils.mock.base_mapping*

and_raises(*error*)

and_return(*return_obj*)

class virttest.unittest_utils.mock.**is_instance_comparator**(*cls*)
Bases: *virttest.unittest_utils.mock.argument_comparator*

is_satisfied_by(*parameter*)

class virttest.unittest_utils.mock.**is_string_comparator**
Bases: *virttest.unittest_utils.mock.argument_comparator*

is_satisfied_by(*parameter*)

class virttest.unittest_utils.mock.**mask_function**(*symbol*, *original_function*, *default_return_val=None*, *record=None*, *playback=None*)
Bases: *virttest.unittest_utils.mock.mock_function*

run_original_function(**args*, ***dargs*)

class virttest.unittest_utils.mock.**mock_class**(*cls*, *name*, *default_ret_val=None*, *record=None*, *playback=None*)
Bases: object

class virttest.unittest_utils.mock.**mock_function**(*symbol*, *default_return_val=None*, *record=None*, *playback=None*)
Bases: object

expect_any_call()
Like expect_call but don't give a hoot what arguments are passed.

expect_call(**args*, ***dargs*)

class virttest.unittest_utils.mock.**mock_god**(*debug=False*, *fail_fast=True*, *ut=None*)
Bases: object

With debug=True, all recorded method calls will be printed as they happen. With fail_fast=True, unexpected calls will immediately cause an exception to be raised. With False, they will be silently recorded and only reported when check_playback() is called.

NONEXISTENT_ATTRIBUTE = <object object>

check_playback()
Report any errors that were encounterd during calls to __method_playback().

create_mock_class(*cls*, *name*, *default_ret_val=None*)
Given something that defines a namespace cls (class, object, module), and a (hopefully unique) name, will create a mock_class object with that name and that possesses all the public attributes of cls. default_ret_val sets the default_ret_val on all methods of the cls mock.

create_mock_class_obj(*cls*, *name*, *default_ret_val=None*)

create_mock_function(*symbol*, *default_return_val=None*)
create a mock_function with name symbol and default return value of default_ret_val.

mock_io()
Mocks and saves the stdout & stderr output

mock_up(*obj*, *name*, *default_ret_val=None*)
Given an object (class instance or module) and a registration name, then replace all its methods with mock function objects (passing the original functions to the mock functions).

set_fail_fast(*fail_fast*)

stub_class(*namespace*, *symbol*)

stub_class_method(*cls*, *symbol*)

stub_function(*namespace*, *symbol*)

stub_function_to_return(*namespace*, *symbol*, *object_to_return*)
Stub out a function with one that always returns a fixed value.

Parameters

- **namespace** – The namespace containing the function to stub out.
- **symbol** – The attribute within the namespace to stub out.
- **object_to_return** – The value that the stub should return whenever it is called.

stub_with(*namespace*, *symbol*, *new_attribute*)

unmock_io()
Restores the stdout & stderr, and returns both output strings

unstub(*namespace*, *symbol*)

unstub_all()

class virttest.unittest_utils.mock.**regex_comparator**(*pattern*, *flags=0*)
Bases: *virttest.unittest_utils.mock.argument_comparator*

is_satisfied_by(*parameter*)

Module contents

virttest.unittests package

Subpackages

virttest.unittests.libvirt_xml package

Submodules

virttest.unittests.libvirt_xml.test_network_xml module

virttest.unittests.libvirt_xml.test_vm_xml module

Module contents

Submodules

virttest.unittests.test_utils_kernel_module module

virttest.unittests.test_utils_misc module

virttest.unittests.test_utils_test__init__ module

virttest.unittests.test_utils_zchannels module

virttest.unittests.test_utils_zcrypt module

Module contents

virttest.utils_libvirt package

Submodules

virttest.utils_libvirt.libvirt_bios module

Libvirt BIOS related utilities.

copyright 2022 Red Hat Inc.

virttest.utils_libvirt.libvirt_bios.**remove_bootconfig_items_from_vmos**(*osxml*)
Remove efi firmware attribute and loader/nvram elements

Parameters **osxml** – VMOSXML object

Returns VMOSXML, the updated object

virttest.utils_libvirt.libvirt_ceph_utils module

virttest.utils_libvirt.libvirt_config module

virttest.utils_libvirt.libvirt_cpu module

virttest.utils_libvirt.libvirt_disk module

virttest.utils_libvirt.libvirt_embedded_qemu module

virttest.utils_libvirt.libvirt_keywrap module

virttest.utils_libvirt.libvirt_memory module

virttest.utils_libvirt.libvirt_misc module

Virtualization test - utility functions for libvirt

copyright 2021 Red Hat Inc.

virttest.utils_libvirt.libvirt_misc.**convert_to_dict**(*content*, *pattern='(\\d+)+(\\S+)'*)
Put the content into a dict according to the pattern.

Parameters

- **content** – str, the string to be parsed
- **pattern** – str, regex for parsing the command output

Returns dict, the dict contains matched result

virttest.utils_test.libvirt_device_utils module

virttest.utils_test.libvirt_domjobinfo module

Module contents

virttest.utils_windows package

Submodules

virttest.utils_windows.drive module

virttest.utils_windows.system module

Windows system utilities

virttest.utils_windows.system.**file_exists**(*session*, *filename*)
Check if a file exists.

Parameters

- **session** – Session object.
- **filename** – File name with full path.

Returns bool value: True if file exists.

virttest.utils_windows.system.**os_arch**(*session*)
Get Windows OS architecture.

Parameters **session** – Session object.

Returns Windows OS architecture.

virttest.utils_windows.system.**product_name**(*session*)
Get Windows product name.

Parameters **session** – Session object.

Returns Windows product name.

virttest.utils_windows.system.**version**(*session*)
Get Windows version.

Parameters **session** – Session object.

Returns Windows version.

virttest.utils_windows.virtio_win module

virttest.utils_windows.wmic module

Windows WMIC utilities

virttest.utils_windows.wmic.**is_noinstance**(*data*)
Check if the given WMIC data contains instance(s).

Parameters **data** – The given WMIC data.

Returns True if no instance(s), else False.

virttest.utils_windows.wmic.**make_query**(*cmd*, *cond=None*, *props=None*, *get_swch=None*, *gbl_swch=None*)

Make a WMIC query command. The command pattern is:

wmic [GBL_SWCH] CMD [where COND] get [PROPS] [GET_SWCH]

Parameters

- **cmd** – WMIC command.
- **cond** – Query condition to be appended to *where*.
- **props** – Properties to be get.
- **get_swch** – Local switch of *get*.
- **gbl_swch** – Global switch.

Returns Query command.

virttest.utils_windows.wmic.**parse_list**(*data*)

Parse the given list format WMIC data.

Parameters **data** – The given WMIC data.

Returns Formatted data.

Module contents

virttest.vt_utils package

Submodules

virttest.vt_utils.cpu module

virttest.vt_utils.sys_time module

Module contents

Submodules

virttest.RFBDes module

virttest.arch module

virttest.asset module

virttest.base_installer module

virttest.bootstrap module

virttest.build_helper module

virttest.cartesian_config module

Cartesian configuration format file parser.

Filter syntax:

- `,` means `OR`
- `..` means `AND`
- `.` means `IMMEDIATELY-FOLLOWED-BY`
- `(xx=yy)` where `xx=VARIANT_NAME` and `yy=VARIANT_VALUE`

Example:

```
qcow2..(guest_os=Fedora).14, RHEL.6..raw..boot, smp2..qcow2..migrate..ide
```

means match all dicts whose names have:

```
(qcow2 AND ((guest_os=Fedora) IMMEDIATELY-FOLLOWED-BY 14)) OR
((RHEL IMMEDIATELY-FOLLOWED-BY 6) AND raw AND boot) OR
(smp2 AND qcow2 AND migrate AND ide)
```

Note:

- `qcow2..Fedora.14` is equivalent to `Fedora.14..qcow2`.
- `qcow2..Fedora.14` is not equivalent to `qcow2..14.Fedora`.
- `ide, scsi` is equivalent to `scsi, ide`.

Filters can be used in 3 ways:

```
only <filter>
no <filter>
<filter>:
```

The last one starts a conditional block.

Formal definition: Regexp come from python. They're not deterministic, but more readable for people. Spaces between terminals and nonterminals are only for better reading of definitions.

The base of the definitions come verbatim as follows:

```
E = {\n, #, :, "-", =, +=, <=, ~=, ?=, ?+=, ?<=, !, < , del, @, variants, include,␣
↪only, no, name, value}

N = {S, DEL, FILTER, FILTER_NAME, FILTER_GROUP, PN_FILTER_GROUP, STAT, VARIANT, VAR-
↪TYPE, VAR-NAME, VAR-NAME-F, VAR, COMMENT, TEXT, DEPS, DEPS-NAME-F, META-DATA,␣
↪IDENTIFIER}``


I = I^n | n in N              // indentation from start of line
                              // where n is indentation length.
I = I^n+x | n,x in N          // indentation with shift

start symbol = S
end symbol = eps

S -> I^0+x STATV | eps
```

(continues on next page)

(continued from previous page)

```
I^n    STATV
I^n    STATV

I^n STATV -> I^n STATV \n I^n STATV | I^n STAT | I^n variants VARIANT
I^n STAT -> I^n STAT \n I^n STAT | I^n COMMENT | I^n include INC
I^n STAT -> I^n del DEL | I^n FILTER

DEL -> name \n

I^n STAT -> I^n name = VALUE | I^n name += VALUE | I^n name <= VALUE | I^n name ~=␣
↪VALUE
I^n STAT -> I^n name ?= VALUE | I^n name ?+= VALUE | I^n name ?<= VALUE

VALUE -> TEXT \n | 'TEXT' \n | "TEXT" \n

COMMENT_BLOCK -> #TEXT | //TEXT
COMMENT ->  COMMENT_BLOCK\n
COMMENT ->  COMMENT_BLOCK\n

TEXT = [^\n] TEXT            //python format regexp

I^n    variants VAR #comments:             add possibility for comment
I^n+x       VAR-NAME: DEPS
I^n+x+x2        STATV
I^n         VAR-NAME:

IDENTIFIER -> [A-Za-z0-9][A-Za-z0-9_-]*

VARIANT -> VAR COMMENT_BLOCK\n I^n+x VAR-NAME
VAR -> VAR-TYPE: | VAR-TYPE META-DATA: | :         // Named | unnamed variant

VAR-TYPE -> IDENTIFIER

variants _name_ [xxx] [zzz=yyy] [uuu]:

META-DATA -> [IDENTIFIER] | [IDENTIFIER=TEXT] | META-DATA META-DATA

I^n VAR-NAME -> I^n VAR-NAME \n I^n VAR-NAME | I^n VAR-NAME-N \n I^n+x STATV
VAR-NAME-N -> - @VAR-NAME-F: DEPS | - VAR-NAME-F: DEPS
VAR-NAME-F -> [a-zA-Z0-9\._-]+                  // Python regexp

DEPS -> DEPS-NAME-F | DEPS-NAME-F,DEPS
DEPS-NAME-F -> [a-zA-Z0-9\._- ]+                // Python regexp

INC -> name \n


FILTER_GROUP: STAT
    STAT

I^n STAT -> I^n PN_FILTER_GROUP | I^n ! PN_FILTER_GROUP

PN_FILTER_GROUP -> FILTER_GROUP: \n I^n+x STAT
PN_FILTER_GROUP -> FILTER_GROUP: STAT \n I^n+x STAT

only FILTER_GROUP
no FILTER_GROUP
```

(continues on next page)

(continued from previous page)

```
FILTER -> only FILTER_GROUP \n | no FILTER_GROUP \n

FILTER_GROUP -> FILTER_NAME
FILTER_GROUP -> FILTER_GROUP..FILTER_GROUP
FILTER_GROUP -> FILTER_GROUP,FILTER_GROUP

FILTER_NAME -> FILTER_NAME.FILTER_NAME
FILTER_NAME -> VAR-NAME-F | (VAR-NAME-F=VAR-NAME-F)
```

copyright Red Hat 2008-2013

class virttest.cartesian_config.**BlockFilter**(*blocked*)
Bases: object

apply_to_dict(*d*)

blocked

class virttest.cartesian_config.**Condition**(*lfilter*, *line*)
Bases: *virttest.cartesian_config.NoFilter*

content

class virttest.cartesian_config.**FileReader**(*filename*)
Bases: *virttest.cartesian_config.StrReader*

Preprocess an input file for easy reading.

Initialize the reader.

Parse filename The name of the input file.

class virttest.cartesian_config.**Filter**(*lfilter*)
Bases: object

filter

match(*ctx*, *ctx_set*)

might_match(*ctx*, *ctx_set*, *descendant_labels*)

class virttest.cartesian_config.**JoinFilter**(*lfilter*, *line*)
Bases: *virttest.cartesian_config.NoOnlyFilter*

class virttest.cartesian_config.**LAnd**
Bases: *virttest.cartesian_config.Token*

identifier = '..'

class virttest.cartesian_config.**LAppend**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

identifier = '+='

class virttest.cartesian_config.**LApplyPreDict**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

identifier = 'apply_pre_dict'

set_operands(*name*, *value*)

class virttest.cartesian_config.**LCoc**
Bases: *virttest.cartesian_config.Token*

identifier = '.'

class virttest.cartesian_config.**LColon**
Bases: *virttest.cartesian_config.Token*

identifier = ':'

class virttest.cartesian_config.**LComa**
Bases: *virttest.cartesian_config.Token*

identifier = ','

class virttest.cartesian_config.**LCond**
Bases: *virttest.cartesian_config.Token*

identifier = ''

class virttest.cartesian_config.**LDefault**
Bases: *virttest.cartesian_config.Token*

identifier = '@'

class virttest.cartesian_config.**LDel**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

identifier = 'del'

class virttest.cartesian_config.**LDot**
Bases: *virttest.cartesian_config.Token*

identifier = '.'

class virttest.cartesian_config.**LEndBlock**(*length*)
Bases: *virttest.cartesian_config.LIndent*

class virttest.cartesian_config.**LEndL**
Bases: *virttest.cartesian_config.Token*

identifier = 'endl'

class virttest.cartesian_config.**LIdentifier**
Bases: str

checkAlpha()
Check if string contain only chars

checkChar(*chars*)

checkCharAlpha(*chars*)
Check if string contain only chars

checkCharAlphaNum(*chars*)
Check if string contain only chars

checkCharNumeric(*chars*)
Check if string contain only chars

checkNumbers()
Check if string contain only chars

identifier = 'Identifier re([A-Za-z0-9][A-Za-z0-9_-]*)'

class virttest.cartesian_config.**LInclude**
Bases: *virttest.cartesian_config.Token*

identifier = 'include'

class virttest.cartesian_config.**LIndent**(*length*)
Bases: *virttest.cartesian_config.Token*

identifier = 'indent'

length

class virttest.cartesian_config.**LJoin**
Bases: *virttest.cartesian_config.Token*

identifier = 'join'

class virttest.cartesian_config.**LLBracket**
Bases: *virttest.cartesian_config.Token*

identifier = '['

class virttest.cartesian_config.**LLRBracket**
Bases: *virttest.cartesian_config.Token*

identifier = '('

class virttest.cartesian_config.**LLazySet**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

identifier = '~='

class virttest.cartesian_config.**LNo**
Bases: *virttest.cartesian_config.Token*

identifier = 'no'

class virttest.cartesian_config.**LNotCond**
Bases: *virttest.cartesian_config.Token*

identifier = '!'

class virttest.cartesian_config.**LOnly**
Bases: *virttest.cartesian_config.Token*

identifier = 'only'

class virttest.cartesian_config.**LOperators**
Bases: *virttest.cartesian_config.Token*

function = None

identifier = ''

name

set_operands(*name*, *value*)

value

class virttest.cartesian_config.**LOr**
Bases: *virttest.cartesian_config.Token*

identifier = ','

class virttest.cartesian_config.**LPrepend**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

identifier = '<='

class virttest.cartesian_config.**LRBracket**
Bases: *virttest.cartesian_config.Token*

identifier = ']'

class virttest.cartesian_config.**LRRBracket**
Bases: *virttest.cartesian_config.Token*

identifier = ')'

class virttest.cartesian_config.**LRegExpAppend**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

identifier = '?+='

class virttest.cartesian_config.**LRegExpPrepend**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

identifier = '?<='

class virttest.cartesian_config.**LRegExpSet**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

identifier = '?='

class virttest.cartesian_config.**LRegExpStart**
Bases: *virttest.cartesian_config.Token*

identifier = '${'

class virttest.cartesian_config.**LRegExpStop**
Bases: *virttest.cartesian_config.Token*

identifier = '}'

class virttest.cartesian_config.**LSet**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

Parameters **d** – Dictionary for apply value

identifier = '='

class virttest.cartesian_config.**LString**
Bases: *virttest.cartesian_config.LIdentifier*

identifier = 'String re(.+)'

class virttest.cartesian_config.**LSuffix**
Bases: *virttest.cartesian_config.Token*

identifier = 'suffix'

class virttest.cartesian_config.**LUpdateFileMap**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

dest

identifier = 'update_file_map'

set_operands(*filename*, *name*, *dest='_name_map_file'*)

shortname

class virttest.cartesian_config.**LVariant**
Bases: *virttest.cartesian_config.Token*

identifier = '-'

class virttest.cartesian_config.**LVariants**
Bases: *virttest.cartesian_config.Token*

identifier = 'variants'

class virttest.cartesian_config.**LWhite**
Bases: *virttest.cartesian_config.LIdentifier*

identifier = 'WhiteSpace re(\\s)'

class virttest.cartesian_config.**Label**(*name*, *next_name=None*)
Bases: object

hash_name()

hash_val

hash_var

hash_variant()

long_name

name

var_name

class virttest.cartesian_config.**Lexer**(*reader*)
Bases: object

check_token(*token*, *lType*)

flush_until(*end_tokens=None*)

get_lexer()

get_next_check(*lType*)

get_next_check_nw(*lType*)

get_until(*end_tokens=None*)

get_until_check(*lType*, *end_tokens=None*)
Read tokens from iterator until get end_tokens or type of token not match ltype

Parameters

- **lType** – List of allowed tokens
- **end_tokens** – List of tokens for end reading

Returns List of readed tokens.

get_until_gen(*end_tokens=None*)

get_until_no_white(*end_tokens=None*)
Read tokens from iterator until get one of end_tokens and strip LWhite

Parameters **end_tokens** – List of tokens for end reading

Returns List of readed tokens.

match(*line*, *pos*)

rest_line()

rest_line_as_LString()

rest_line_gen()

rest_line_no_white()

set_fast()

set_prev_indent(*prev_indent*)

set_strict()

exception virttest.cartesian_config.**LexerError**(*msg*, *line=None*, *filename=None*, *linenum=None*)
Bases: *virttest.cartesian_config.ParserError*

exception virttest.cartesian_config.**MissingIncludeError**(*line*, *filename*, *linenum*)
Bases: exceptions.Exception

class virttest.cartesian_config.**NegativeCondition**(*lfilter*, *line*)
Bases: *virttest.cartesian_config.OnlyFilter*

content

class virttest.cartesian_config.**NoFilter**(*lfilter*, *line*)
Bases: *virttest.cartesian_config.NoOnlyFilter*

is_irrelevant(*ctx*, *ctx_set*, *descendant_labels*)

might_pass(*failed_ctx*, *failed_ctx_set*, *ctx*, *ctx_set*, *descendant_labels*)

requires_action(*ctx*, *ctx_set*, *descendant_labels*)

class virttest.cartesian_config.**NoOnlyFilter**(*lfilter*, *line*)
Bases: *virttest.cartesian_config.Filter*

line

class virttest.cartesian_config.**Node**
Bases: object

append_to_shortname

children

content

default

dep

dump(*indent*, *recurse=False*)

failed_cases

filename

labels

name

q_dict

var_name

class virttest.cartesian_config.**OnlyFilter**(*lfilter*, *line*)
Bases: *virttest.cartesian_config.NoOnlyFilter*

is_irrelevant(*ctx*, *ctx_set*, *descendant_labels*)

might_pass(*failed_ctx*, *failed_ctx_set*, *ctx*, *ctx_set*, *descendant_labels*)

requires_action(*ctx*, *ctx_set*, *descendant_labels*)

class virttest.cartesian_config.**Parser**(*filename=None*, *defaults=False*, *expand_defaults=[]*, *debug=False*)
Bases: object

assign(*key*, *value*)
Apply an assignment programatically and keep track of it.

Equivalent to parse a "key = value" line.

Parameters **variant** – String with the variant name.

get_dicts(*node=None*, *ctx=[]*, *content=[]*, *shortname=[]*, *dep=[]*)
Process 'join' entry, unpack join filter for node.

Parameters

- **ctx** – node labels/names
- **content** – previous content in plain

Returns dictionary

1) **join filter_1 filter_2 ....**

   multiplies all dictionaries as: all_variants_match_filter_1 * all_variants_match_filter_2 * ....

2) **join only_one_filter** == only only_one_filter

3) **join filter_1 filter_1**

   also works and transforms to: all_variants_match_filter_1 * all_variants_match_filter_1

   Example: join a join a

   Transforms into: join a a

get_dicts_plain(*node=None*, *ctx=[]*, *content=[]*, *shortname=[]*, *dep=[]*)
Generate dictionaries from the code parsed so far. This should be called after parsing something.

Returns A dict generator.

mk_name(*n1*, *n2*)
Make name for test. Case: two dics were merged

multiply_join(*onlys*, *node=None*, *ctx=[]*, *content=[]*, *shortname=[]*, *dep=[]*)
Multiply all joins. Return dictionaries one by one Each 'join' is the same as 'only' filter This functions is supposed to be a generator, recursive generator

no_filter(*variant*)
Apply a no filter programatically and keep track of it.

Equivalent to parse a "no variant" line.

Parameters **variant** – String with the variant name.

only_filter(*variant*)
Apply a only filter programatically and keep track of it.

Equivalent to parse a "only variant" line.

Parameters **variant** – String with the variant name.

parse_file(*filename*)
Parse a file.

Parameters **filename** – Path of the configuration file.

parse_string(*s*)
Parse a string.

Parameters **s** – String to parse.

exception virttest.cartesian_config.**ParserError**(*msg*, *line=None*, *filename=None*, *linenum=None*)
Bases: exceptions.Exception

class virttest.cartesian_config.**StrReader**(*s*)
Bases: object

Preprocess an input string for easy reading.

Initialize the reader.

Parameters **s** – The string to parse.

get_next_line(*prev_indent*)
Get the next line in the current block.

Parameters **prev_indent** – The indentation level of the previous block.

Returns (line, indent, linenum), where indent is the line's indentation level. If no line is available, (None, -1, -1) is returned.

set_next_line(*line*, *indent*, *linenum*)
Make the next call to get_next_line() return the given line instead of the real next line.

class virttest.cartesian_config.**Suffix**
Bases: *virttest.cartesian_config.LOperators*

apply_to_dict(*d*)

identifier = 'apply_suffix'

class virttest.cartesian_config.**Token**
Bases: object

identifier = ''

virttest.cartesian_config.**apply_predict**(*lexer*, *node*, *pre_dict*)

virttest.cartesian_config.**cmd_tokens**(*tokens1*, *tokens2*)

virttest.cartesian_config.**compare_string**(*str1*, *str2*)
Compare two int string and return -1, 0, 1. It can compare two memory value even in sufix

Parameters

- **str1** – The first string
- **str2** – The second string

Return Return -1, when str1< str2 0, when str1 = str2 1, when str1> str2

virttest.cartesian_config.**convert_data_size**(*size*, *default_sufix='B'*)
Convert data size from human readable units to an int of arbitrary size.

Parameters

- **size** – Human readable data size representation (string).
- **default_sufix** – Default sufix used to represent data.

Returns Int with data size in the appropriate order of magnitude.

virttest.cartesian_config.**next_nw**(*gener*)

virttest.cartesian_config.**parse_filter**(*lexer*, *tokens*)

Returns Parsed filter

virttest.cartesian_config.**postfix_parse**(*dic*)

virttest.cartesian_config.**print_dicts**(*options*, *dicts*)

virttest.cartesian_config.**print_dicts_default**(*options*, *dicts*)
Print dictionaries in the default mode

virttest.cartesian_config.**print_dicts_repr**(*options*, *dicts*)

virttest.ceph module

virttest.compat module

virttest.cpu module

virttest.curl module

virttest.data_dir module

virttest.defaults module

virttest.defaults.**get_default_guest_os_info**()
Gets the default asset and variant information TODO: Check for the ARCH and choose corresponding default asset

virttest.env_process module

virttest.error_context module

virttest.error_context.**format_error**()

virttest.error_context.**context_aware**(*fn*)
A decorator that must be applied to functions that call context().

virttest.error_context.**context**(*s=''*, *log=None*)
Set the context for the currently executing function and optionally log it.

Parameters

- **s** – A string. If not provided, the context for the current function will be cleared.
- **log** – A logging function to pass the context message to. If None, no function will be called.

virttest.error_context.**get_context**()
Return the current context (or None if none is defined).

virttest.error_context.**exception_context**(*e*)
Return the context of a given exception (or None if none is defined).

virttest.error_event module

Global background error event bus for vt test object. This aims to share error event bus with asynchronous tasks launched by avocado-vt, so that those tasks that have no reference to test could error the test.

IMPORTANT: only for internal use inside avocado-vt.

class virttest.error_event.**EventBus**
Bases: object

Event Bus.

Create the error event bus with queue.Queue.

clear()
Clear all events in the event bus.

get(*block=True*, *timeout=None*)
Remove and return an event from the event bus.

get_all()
Remove and return a list of all events from the event bus.

put(*event*, *block=True*, *timeout=None*)
Put an event into the event bus.

virttest.funcatexit module

virttest.gluster module

virttest.graphical_console module

virttest.guest_agent module

virttest.http_server module

virttest.installer module

virttest.ip_sniffing module

virttest.iscsi module

virttest.kernel_interface module

virttest.libvirt_cgroup module

virttest.libvirt_installer module

virttest.libvirt_remote module

virttest.libvirt_storage module

virttest.libvirt_version module

virttest.libvirt_vm module

virttest.libvirtd_decorator module

virttest.logging_manager module

class virttest.logging_manager.**LoggingFile**(*prefix=''*, *level=10*, *logger=<logging.Logger object>*)

Bases: object

File-like object that will receive messages pass them to the logging infrastructure in an appropriate way.

:param prefix - The prefix for each line logged by this object.

flush()

isatty()

write(*data*)

" Writes data only if it constitutes a whole line. If it's not the case, store it in a buffer and wait until we have a complete line. :param data - Raw data (a string) that will be processed.

writelines(*lines*)

" Writes itertable of lines

Parameters **lines** – An iterable of strings that will be processed.

virttest.logging_manager.**do_not_report_as_logging_caller**(*func*)

Decorator to annotate functions we will tell logging not to log.

virttest.lvm module

virttest.lvsb module

virttest.lvsb_base module

virttest.lvsbs module

virttest.migration module

virttest.migration_template module

virttest.nbd module

virttest.nfs module

virttest.nvme module

virttest.openvswitch module

virttest.ovirt module

virttest.ovs_utils module

virttest.postprocess_iozone module

virttest.ppm_utils module

Utility functions to deal with ppm (qemu screendump format) files.

copyright Red Hat 2008-2009

virttest.ppm_utils.**add_timestamp**(*image*, *timestamp*, *margin=2*)
Return an image object with timestamp bar added at the bottom.

param image: pillow image object param timestamp: timestamp in seconds since the Epoch param margin: timestamp margin, default is 2

virttest.ppm_utils.**cal_hamming_distance**(*h1*, *h2*)
Calculate the hamming distance

virttest.ppm_utils.**find_id_for_screendump**(*md5sum*, *data_dir*)
Search dir for a PPM file whose name ends with md5sum.

Parameters

- **md5sum** – md5 sum string
- **dir** – Directory that holds the PPM files.

Returns The file's basename without any preceding path, e.g. `20080101_120000_d41d8cd98f00b204e9800998ecf8427e.ppm`

virttest.ppm_utils.**generate_id_for_screendump**(*md5sum*, *data_dir*)
Generate a unique filename using the given MD5 sum.

Returns Only the file basename, without any preceding path. The filename consists of the current date and time, the MD5 sum and a `.ppm` extension, e.g. `20080101_120000_d41d8cd98f00b204e9800998ecf8427e.ppm`.

virttest.ppm_utils.**get_data_dir**(*steps_filename*)
Return the data dir of the given steps filename.

virttest.ppm_utils.**get_region_md5sum**(*width*, *height*, *data*, *x1*, *y1*, *dx*, *dy*, *cropped_image_filename=None*)
Return the md5sum of a cropped region.

Parameters

- **width** – Original image width
- **height** – Original image height
- **data** – Image data
- **x1** – Desired x coord of the cropped region
- **y1** – Desired y coord of the cropped region
- **dx** – Desired width of the cropped region
- **dy** – Desired height of the cropped region

- **cropped_image_filename** – if not None, write the resulting cropped image to a file with this name

virttest.ppm_utils.**have_similar_img**(*base_img*, *comp_img_path*, *threshold=10*)
Check whether comp_img_path have a image looks like base_img.

virttest.ppm_utils.**image_average_hash**(*image*, *img_wd=8*, *img_ht=8*)
Resize and convert the image, then get image data as sequence object, calculate the average hash :param image: an image path or an opened image object

virttest.ppm_utils.**image_comparison**(*width*, *height*, *data1*, *data2*)
Generate a green-red comparison image from two given images.

Parameters

- **width** – Width of both images
- **height** – Height of both images
- **data1** – Data of first image
- **data2** – Data of second image

Returns A 3-element tuple containing the width, height and data of the generated comparison image.

Note Input images must be the same size.

virttest.ppm_utils.**image_crop**(*width*, *height*, *data*, *x1*, *y1*, *dx*, *dy*)
Crop an image.

Parameters

- **width** – Original image width
- **height** – Original image height
- **data** – Image data
- **x1** – Desired x coordinate of the cropped region
- **y1** – Desired y coordinate of the cropped region
- **dx** – Desired width of the cropped region
- **dy** – Desired height of the cropped region

Returns A 3-tuple containing the width, height and data of the cropped image.

virttest.ppm_utils.**image_crop_save**(*image*, *new_image*, *box=None*)
Crop an image and save it to a new image.

Parameters

- **image** – Full path of the original image
- **new_image** – Full path of the cropped image
- **box** – A 4-tuple defining the left, upper, right, and lower pixel coordinate.

Returns True if crop and save image succeed

virttest.ppm_utils.**image_fuzzy_compare**(*width*, *height*, *data1*, *data2*)
Return the degree of equality of two given images.

Parameters

- **width** – Width of both images
- **height** – Height of both images

- **data1** – Data of first image
- **data2** – Data of second image

Returns Ratio equal_pixel_count / total_pixel_count.

Note Input images must be the same size.

virttest.ppm_utils.**image_histogram_compare**(*image_a*, *image_b*, *size=(0, 0)*)
Compare the histogram of two images and return similar degree.

Parameters

- **image_a** – Full path of the first image
- **image_b** – Full path of the second image
- **size** – Convert image to size(width, height), and if size=(0, 0), the function will convert the big size image align with the small one.

virttest.ppm_utils.**image_md5sum**(*width*, *height*, *data*)
Return the md5sum of an image.

Parameters

- **width** – PPM file width
- **height** – PPM file height
- **data** – PPM file data

virttest.ppm_utils.**image_read_from_ppm_file**(*filename*)
Read a PPM image.

Returns A 3 element tuple containing the width, height and data of the image.

virttest.ppm_utils.**image_size**(*image*)
Return image's size as a tuple (width, height).

Parameters **image** – image file.

virttest.ppm_utils.**image_verify_ppm_file**(*filename*)
Verify the validity of a PPM file.

Parameters **filename** – Path of the file being verified.

Returns True if filename is a valid PPM image file. This function reads only the first few bytes of the file so it should be rather fast.

virttest.ppm_utils.**image_write_to_ppm_file**(*filename*, *width*, *height*, *data*)
Write a PPM image with the given width, height and data.

Parameters

- **filename** – PPM file path
- **width** – PPM file width (pixels)
- **height** – PPM file height (pixels)

virttest.ppm_utils.**img_ham_distance**(*base_img*, *comp_img*)
Calculate two images hamming distance

virttest.ppm_utils.**img_similar**(*base_img*, *comp_img*, *threshold=10*)
check whether two images are similar by hamming distance

virttest.propcan module

Class which allows property and dict-like access to a fixed set of instance attributes. Attributes are locked by __slots__, however accessor methods may be created/removed on instances, or defined by the subclass. An INITIALIZED attribute is provided to signal completion of __init__() for use by accessor methods (i.e. so they know when __init__ may be setting values).

Subclasses must define a __slots__ class attribute containing the list of attribute names to reserve. All additional subclass descendents must explicitly copy __slots__ from the parent in their definition.

Users of subclass instances are expected to get/set/del attributes only via the standard object or dict-like interface. i.e.

instance.attribute = whatever or instance['attribute'] = whatever

Internally, methods are free to call the accessor methods. Only accessor methods should use the special __dict_*__() and __super_*__() methods. These are there to allow convenient access to the internal dictionary values and subclass-defined attributes (such as __slots__).

example:

```
class A(PropCan):
   # Class with *attributes*
    __slots__ = ('a', 'b')
    # 'a' has defined a set/get/del by definition of method with prefix
    #     set_a, get_a, del_a
    # 'b' doesn't have defined set/get/del then classic set/get/del will be
    #     called instead.


    def __init__(self, a=1, b='b'):
       super(A, self).__init__(a, b)


    def set_a(self, value)
        # If is_instance(obj, A) then obj.a = "val" call this method.
        self.__dict_set__("a", value)


    def get_a(self, value)
        # If is_instance(obj, A) then xx = obj.a call this method.
        return self.__dict_get__("a")


    def del_a(self, value)
        # If is_instance(obj, A) then del obj.a call this method.
        self.__dict_del__("a")


class B(PropCan):
   # Class without *attributes*
   # ***** Even if class doesn't have attributes there should be
   # defined __slots__ = []. Because it is preferred by new style of class.
   # *****
    __slots__ = []


    def __init__(self):
       super(B, self).__init__()
```

class virttest.propcan.**PropCan**(**args*, ***dargs*)
Bases: *virttest.propcan.PropCanBase*

Special value handling on retrieval of None values

Initialize contents directly or by way of accessors

Parameters

- **args** – Initial values for __slots__ keys, same as dict.
- **dargs** – Initial values for __slots__ keys, same as dict.

has_key(*key*)
D.__contains__(k) -> True if D has a key k, else False

items() → list of D's (key, value) pairs, as 2-tuples

keys() → list of D's keys

set_if_none(*key*, *value*)
Set the value of key, only if it's not set or None

set_if_value_not_none(*key*, *value*)
Set the value of key, only if value is not None

values() → list of D's values

class virttest.propcan.**PropCanBase**(**args*, ***dargs*)
Bases: dict, *virttest.propcan.PropCanInternal*

Objects with optional accessor methods and dict-like access to fixed set of keys

Initialize contents directly or by way of accessors

Parameters

- **args** – Initial values for __slots__ keys, same as dict.
- **dargs** – Initial values for __slots__ keys, same as dict.

INITIALIZED = False

copy()
Copy properties by value, not by reference.

update(*other=None*, *excpt=<type 'exceptions.AttributeError'>*, ***kwargs*)
Update properties in __all_slots__ with another dict.

class virttest.propcan.**PropCanInternal**
Bases: object

Semi-private methods for use only by PropCanBase subclasses (NOT instances)

class virttest.propcan.**classproperty**
Bases: property

virttest.qemu_capabilities module

Module for defining the capabilities of vm.

Available class: - Flags: Enumerate of the flags of VM capabilities. - Capabilities: Representation of VM capabilities.

class virttest.qemu_capabilities.**Capabilities**
Bases: object

Representation of VM capabilities.

clear_flag(*flag*)
Clear the flag.

Parameters **flag** (Flags) – The name of flag.

set_flag(*flag*)
Set the flag.

Parameters **flag** (Flags) – The name of flag.

class virttest.qemu_capabilities.**Flags**
Bases: object

Enumerate the flags of VM capabilities.

BLOCKDEV = 0

FLOPPY_DEVICE = 8

INCOMING_DEFER = 3

MACHINE_MEMORY_BACKEND = 4

MIGRATION_PARAMS = 5

SEV_GUEST = 6

SMP_CLUSTERS = 2

SMP_DIES = 1

TDX_GUEST = 7

class virttest.qemu_capabilities.**MigrationParams**
Bases: object

Enumerate migration parameters.

DOWNTIME_LIMIT = 9

MAX_BANDWIDTH = 10

XBZRLE_CACHE_SIZE = 11

virttest.qemu_installer module

virttest.qemu_io module

virttest.qemu_migration module

Interface for QEMU migration.

virttest.qemu_migration.**set_cache_size**(*vm*, *value*)
Set cache size for migration.

Parameters

- **vm** – VM object.
- **value** – Cache size to set.

Returns Output of command.

virttest.qemu_migration.**set_downtime**(*vm*, *value*)
Set maximum tolerated downtime for migration.

Parameters

- **vm** – VM object.
- **value** – Maximum downtime in seconds.

Returns Output of command.

virttest.qemu_migration.**set_speed**(*vm*, *value*)
Set maximum speed for migration.

Parameters

- **vm** – VM object.
- **value** – Speed in bytes/sec.

Returns Output of command.

virttest.qemu_monitor module

virttest.qemu_qtree module

virttest.qemu_storage module

virttest.qemu_virtio_port module

virttest.qemu_vm module

virttest.remote module

virttest.remote_build module

virttest.scan_autotest_results module

Program that parses the autotest results and return a nicely printed final test result.

copyright Red Hat 2008-2009

virttest.scan_autotest_results.**main**(*resfiles*)

virttest.scan_autotest_results.**parse_results**(*text*)
Parse text containing Autotest results.

Returns A list of result 4-tuples.

virttest.scan_autotest_results.**print_result**(*result*, *name_width*)
Nicely print a single Autotest result.

Parameters

- **result** – a 4-tuple
- **name_width** – test name maximum width

virttest.scheduler module

virttest.ssh_key module

virttest.standalone_test module

virttest.step_editor module

virttest.storage module

virttest.storage_ssh module

virttest.syslog_server module

class virttest.syslog_server.**RequestHandler**(*request*, *client_address*, *server*)

Bases: SocketServer.BaseRequestHandler

A request handler that relays all received messages as DEBUG

FACILITY_NAMES = {0: 'kern', 1: 'user', 2: 'mail', 3: 'daemon', 4: 'security', 5:

LOG_ALERT = 1

LOG_AUTH = 4

LOG_AUTHPRIV = 10

LOG_CRIT = 2

LOG_CRON = 9

LOG_DAEMON = 3

LOG_DEBUG = 7

LOG_EMERG = 0

LOG_ERR = 3

LOG_FTP = 11

LOG_INFO = 6

LOG_KERN = 0

LOG_LOCAL0 = 16

LOG_LOCAL1 = 17

LOG_LOCAL2 = 18

LOG_LOCAL3 = 19

LOG_LOCAL4 = 20

LOG_LOCAL5 = 21

LOG_LOCAL6 = 22

LOG_LOCAL7 = 23

LOG_LPR = 6

LOG_MAIL = 2

LOG_NEWS = 7

LOG_NOTICE = 5

LOG_SYSLOG = 5

LOG_USER = 1

LOG_UUCP = 8

LOG_WARNING = 4

PRIORITY_NAMES = {0: 'emerg', 1: 'alert', 2: 'critical', 3: 'err', 4: 'warning',

RECORD_RE = <_sre.SRE_Pattern object>

decodeFacilityPriority(*priority*)
Decode both the facility and priority embedded in a syslog message

Parameters **priority** (*integer*) – an integer with facility and priority encoded

Returns a tuple with two strings

log(*data*, *message_format=None*)
Logs the received message as a DEBUG message

class virttest.syslog_server.**RequestHandlerTcp**(*request*, *client_address*, *server*)
Bases: *virttest.syslog_server.RequestHandler*

handle()
Handles a single request

class virttest.syslog_server.**RequestHandlerUdp**(*request*, *client_address*, *server*)
Bases: *virttest.syslog_server.RequestHandler*

handle()
Handles a single request

class virttest.syslog_server.**SysLogServerTcp**(*address*)
Bases: SocketServer.TCPServer

class virttest.syslog_server.**SysLogServerUdp**(*address*)
Bases: SocketServer.UDPServer

virttest.syslog_server.**get_default_format**()
Returns the current default message format

virttest.syslog_server.**set_default_format**(*message_format*)
Changes the default message format

Parameters **message_format** (*string*) – a message format string with 3 placeholders: facility, priority and message.

virttest.syslog_server.**syslog_server**(*address=''*, *port=514*, *tcp=True*, *terminate_callable=None*)

virttest.utils_backup module

virttest.utils_config module

virttest.utils_conn module

virttest.utils_disk module

virttest.utils_env module

virttest.utils_gdb module

virttest.utils_hotplug module

virttest.utils_iptables module

virttest.utils_kernel_module module

virttest.utils_libguestfs module

virttest.utils_libvirtd module

virttest.utils_linux_modules module

virttest.utils_logfile module

virttest.utils_misc module

virttest.utils_nbd module

virttest.utils_net module

virttest.utils_netperf module

virttest.utils_npiv module

virttest.utils_numeric module

virttest.utils_numeric.**align_value**(*value*, *factor=1024*)

Value will be factored to the mentioned number.

Parameters

- **value** – value to be checked or changed to factor
- **factor** – value used for align

Returns aligned value

virttest.utils_numeric.**format_size_human_readable**(*value*, *binary=False*, *precision='%.2f'*)

Format a number of bytesize to a human readable filesize.

By default,decimal suffixes and base:10**3 will be used. :param binary: use binary suffixes (KiB, MiB) and use 2*10 as base :param precision: format string to specify precision for float

virttest.utils_numeric.**normalize_data_size**(*value_str*, *order_magnitude='M'*, *factor=1024*)

Normalize a data size in one order of magnitude to another.

Parameters

- **value_str** – a string include the data default unit is 'B'
- **order_magnitude** – the magnitude order of result

- **factor** – int, the factor between two relative order of magnitude. Normally could be 1024 or 1000

:return normalized data size string

virttest.utils_package module

virttest.utils_params module

virttest.utils_pyvmomi module

virttest.utils_qemu module

virttest.utils_sasl module

virttest.utils_scheduling module

Virtualization test utility functions.

copyright 2008-2009 Red Hat Inc.

virttest.utils_scheduling.**timeout**(*timeout*)
Timeout decorator, parameter in seconds.

virttest.utils_selinux module

virttest.utils_spice module

virttest.utils_split_daemons module

virttest.utils_sriov module

virttest.utils_stress module

virttest.utils_switchdev module

virttest.utils_sys module

virttest.utils_time module

virttest.utils_v2v module

virttest.utils_vdpa module

virttest.utils_version module

class virttest.utils_version.**VersionInterval**(*interval*)
Bases: object

A class for a Version Interval object.

An interval is a string representation of a mathematical like interval. e.g: "(3,4]", "[3.10.0,)" A verison is a version string.

Examples::

```
>>> verison_interval = VersionInterval("[2.10.1, 2.11.0)")
>>> verison = "2.10.16"
>>> verison in verison_interval
True
>>> verison = "2.13.0"
>>> verison in verison_interval
False
```

virttest.utils_virtio_port module

virttest.utils_vsock module

virttest.utils_zchannels module

virttest.utils_zcrypt module

virttest.vdpa_blk module

virttest.version module

virttest.versionable_class module

class virttest.versionable_class.**Manager**(*name*, *wrapper=None*)

Bases: object

Manager for module.

Parameters

- **name** (*string*) – Name of module.
- **wrapper** – Module dictionary wrapper. Should be None.

factory(*_class*, **args*, ***kargs*)

Create new class with right version of subclasses.

Goes through class structure and search subclasses with right version.

Parameters

- **_class** (*class.*) – Class which should be prepared.
- **args** – Params for _is_right_ver function.

Params kargs Params for _is_right_ver function.

getcls(*cls*, *orig_class*)

Return class correspond class and original class.

Parameters

- **cls** (*class*) – class for which should be found derived alternative.
- **orig_class** (*class*) – Original class

Returns Derived alternative class

Return type class

class virttest.versionable_class.**ModuleWrapper**(*wrapped*)
Bases: object

Wrapper around module.

Necessary for pickling of dynamic class.

Parameters **wrapped** (*Module.*) – module for wrapping.

class virttest.versionable_class.**VersionableClass**
Bases: object

Class used for marking of mutable class.

If this method is invoked it means that something went wrong because this class should be replaced by *Manager* factory.

virttest.versionable_class.**factory**(*orig_cls*, **args*, ***kargs*)
Create class with specific version.

Parameters

- **orig_class** – Class from which should be derived good version.
- **args** – list of parameters for _ir_right_ver

Params kargs dict of named parameters for _ir_right_ver

Returns params specific class.

Return type class

virttest.versionable_class.**isclass**(*obj*)

Parameters **obj** – Object for inspection if obj is class.

Returns true if the object is a class.

virttest.video_maker module

Video Maker transforms screenshots taken during a test into a HTML 5 compatible video, so that one can watch the screen activity of the whole test from inside your own browser.

This relies on generally available multimedia libraries, frameworks and tools.

virttest.video_maker.**get_video_maker_klass**()

virttest.video_maker.**video_maker**(*input_dir*, *output_file*)
Instantiates the encoder and encodes the input dir.

virttest.virsh module

virttest.virt_admin module

virttest.virt_vm module

virttest.vt_console module

virttest.vt_iothread module

virttest.xml_utils module

Utility module standardized on ElementTree 2.6 to minimize dependencies in python 2.4 systems.

Often operations on XML files suggest making a backup copy first is a prudent measure. However, it's easy to loose track of these temporary files and they can quickly leave a mess behind. The TempXMLFile class helps by trying to clean up the temporary file whenever the instance is deleted, goes out of scope, or an exception is thrown.

The XMLBackup class extends the TempXMLFile class by basing its file- like instances off of an automatically created TempXMLFile instead of pointing at the source. Methods are provided for overwriting the backup copy from the source, or restoring the source from the backup. Similar to TempXMLFile, the temporary backup files are automatically removed. Access to the original source is provided by the sourcefilename attribute.

An interface for querying and manipulating XML data is provided by the XMLTreeFile class. Instances of this class are BOTH file-like and ElementTree-like objects. Whether or not they are constructed from a file or a string, the file-like instance always represents a temporary backup copy. Access to the source (even when itself is temporary) is provided by the sourcefilename attribute, and a (closed) file object attribute sourcebackupfile. See the ElementTree documentation for methods provided by that class.

Finally, the TemplateXML class represents XML templates that support dynamic keyword substitution based on a dictionary. Substitution keys in the XML template (string or file) follow the 'bash' variable reference style ($foo or ${bar}). Extension of the parser is possible by subclassing TemplateXML and overriding the ParserClass class attribute. The parser class should be an ElementTree.TreeBuilder class or subclass. Instances of XMLTreeFile are returned by the parse method, which are themselves temporary backups of the parsed content. See the xml_utils_unittest module for examples.

class virttest.xml_utils.**Sub**(***mapping*)
Bases: object

String substituter using string.Template

Initialize substitution mapping.

substitute(*text*)
Use string.safe_substitute on text and return the result

Parameters **text** – string to substitute

class virttest.xml_utils.**TempXMLFile**(*suffix='.xml'*, *prefix='xml_utils_temp_'*, *mode='w+'*)
Bases: _io.FileIO

Temporary XML file auto-removed on instance del / module exit.

Initialize temporary XML file removed on instance destruction.

param: suffix: temporary file's suffix param: prefix: temporary file's prefix param: mode: second parameter to file()/open() param: buffer: third parameter to file()/open()

unlink()
Unconditionally delete file, ignoring related exceptions

class virttest.xml_utils.**TemplateXML**(*xml*, ***mapping*)
Bases: *virttest.xml_utils.XMLTreeFile*

Template-sourced XML ElementTree backed by temporary file.

Initialize from a XML string or filename, and string.template mapping.

Parameters

- **xml** – A filename or string containing XML
- **mapping** – keys/values to feed with XML to string.template

ParserClass
alias of *TemplateXMLTreeBuilder*

parse(*source*, *parser=None*)
Parse source XML file or filename using TemplateXMLTreeBuilder

Parameters

- **source** – XML file or filename
- **parser** – ignored

restore()
Raise an IOError to protect the original template source.

class virttest.xml_utils.**TemplateXMLTreeBuilder**(***mapping*)
Bases: xml.etree.ElementTree.XMLParser, *virttest.xml_utils.Sub*

Resolve XML templates into temporary file-backed ElementTrees

Initialize parser that substitutes keys with values in data

Parameters **mapping** – values to be substituted for ${key} in XML input

BuilderClass
alias of xml.etree.ElementTree.TreeBuilder

feed(*data*)

class virttest.xml_utils.**XMLBackup**(*sourcefilename*)
Bases: *virttest.xml_utils.TempXMLFile*

Backup file copy of XML data, automatically removed on instance destruction.

Initialize a temporary backup from sourcefilename.

backup()
Overwrite temporary backup with contents of original source.

restore()
Overwrite original source with contents of temporary backup

sourcefilename = None

class virttest.xml_utils.**XMLTreeFile**(*xml*)
Bases: xml.etree.ElementTree.ElementTree, *virttest.xml_utils.XMLBackup*

Combination of ElementTree root and auto-cleaned XML backup file.

Initialize from a string or filename containing XML source.

param: xml: A filename or string containing XML

backup()
Overwrite original source from current tree

backup_copy()
Return a copy of instance, including copies of files

create_by_xpath(*xpath*)
Creates all elements in simplistic xpath from root if not exist

find(*path*)

get_element_string(*xpath*, *index=0*)
Returns the string for the element on xpath.

get_parent(*element*, *relative_root=None*)
Return the parent node of an element or None

param: element: Element to retrieve parent of param: relative_root: Search only below this element

get_parent_map(*element=None*)
Return a child to parent mapping dictionary

param: element: Search only below this element

get_xpath(*element*)
Return the XPath string formed from first-match tag names

read(*size: int*) → bytes. read at most size bytes, returned as bytes.
Only makes one system call, so less data may be returned than requested In non-blocking mode, returns None if no data is available. On end-of-file, returns ‘’.

remove(*element*)
Removes a matching subelement.

Parameters **element** – element to be removed.

remove_by_xpath(*xpath*, *remove_all=False*)
Remove an element found by xpath

Parameters **xpath** – element name or path to remove

reroot(*xpath*)
Return a copy of instance, re-rooted onto xpath

restore()
Overwrite and reparse current tree from original source

sourcebackupfile = None

write(*filename=None*, *encoding='unicode'*)
Write current XML tree to filename, or self.name if None.

virttest.yumrepo module

This module implements classes that allow a user to create, enable and disable YUM repositories on the system.

class virttest.yumrepo.**YumRepo**(*name*, *baseurl*, *path=None*)
Bases: object

Represents a YUM repository

The goal of this class is not to give access to all features of a YUM Repository, but to provide a simple way to configure a valid one during a test run.

Sample usage:

```
>>> mainrepo = YumRepo("main", "http://download.project.org/repo",
                       "/etc/yum.repos.d/main.repo")
```

Or to use a default path:

```
>>> mainrepo = YumRepo("main", 'http://download.project.org/repo')
```

And then:

```
>>> mainrepo.save()
```

When it comes to the repo URL, currently there's no support for setting a mirrorlist, only a baseurl.

Initilizes a new YumRepo object

If path is not given, it is assumed to be "$(name)s.repo" at the default YUM repo directory.

Parameters

- **name** – the repository name
- **path** – the full path of the file that defines this repository

remove()
Removes the repo file

render()
Renders the repo file

Yes, we could use ConfigParser for this, but it produces files with spaces between keys and values, which look akward by YUM defaults.

save()
Saves the repo file

Module contents

CHAPTER 8

Cartesian Config Reference

8.1 Cartesian Config

Reference documentation of the cartesian config format.

Contents:

8.1.1 Parameters

The test configuration file is used for controlling the framework by specifying parameters for each test. The parser produces a list of dictionaries (see an explanation of the file format?), each of which specifies the parameters for a single test. Each parameter is used (read) by the test dispatching system, by the pre-processor, by the post-processor, or by the test itself.

Some parameters are required and others are optional.

Most parameters should be specified in the test configuration file by the user. Few parameters are produced automatically by the configuration file parser, so when using the parser, these must not be specified by the user. Recent kvm-autotest support passing some basic parameters through the command line?.

All parameters are strings, except depend?, which is a list of strings. depend? is automatically generated by the parser, so the user probably should not be concerned with it.

You may also want to check the complete *reference documentation* on parameters.

Addressing objects (VMs, images, NICs etc)

Before listing the parameters, it is important to clarify how objects like VMs should be addressed in the test parameters.

For the following example we shall assume that our system accepts a parameter vms? which lists the VM objects to be used in the current test. Typical usage of the parameter would be:

```
vms = vm1 second_vm another_vm
```

This would indicate that our test requires 3 VMs. Let us now assume that a VM object accepts a parameter mem which specifies the amount of memory to give the VM. In order to specify mem for **vm1**, we may write:

```
mem_vm1 = 512
```

and in order to specify it for **second_vm** we may write:

```
mem_second_vm = 1024
```

If we wanted to specify mem for all existing VM objects, we would write:

```
mem = 128
```

However, this would only apply to **another_vm**, because the previous statements, which each specify mem for a single VM, override the statement that specifies mem for all VMs. The order in which these statements are written in a configuration file is not important; statements addressing a single object always override statements addressing all objects.

Let us now further assume that a VM object accepts a parameter images, which lists the disk image objects to be used by the VM. Typical usage of images, with regard to **vm1**, would be:

```
images_vm1 = first_image image2 a_third_image yet_another_image
```

We shall also assume that an image object accepts two parameters: image_name, which specifies the filename of the disk image, and image_size, which specifies the size of the image (e.g. 10G). In order to specify these with regard to **first_image**, which is the first image of **vm1**, we may write:

```
image_name_first_image_vm1 = fc8-32-no-acpi
image_size_first_image_vm1 = 20G
```

Note the order in which the objects are addressed: first the parameter, then the image, then the VM. In order to specify these parameters for all images of **vm1**, we may write:

```
image_name_vm1 = fc8-32
image_size_vm1 = 10G
```

However, these statements would not apply to **first_image** of **vm1**, because the previous statements, which addressed this image specifically, override the statements that address all objects. If we chose to specify these parameters for all images of all VMs, we would write:

```
image_name = fc8-32-something
image_size = 5G
```

However, these would not apply to the images of **vm1**, because previous statements apply specifically to those images.

Parameters used by the test dispatching system

The test dispatching system consists of the control file and the framework's main python module (currently named kvm_runtest_2.py). This system executes the proper test according to the supplied parameters.

Parameter	Effect/meaning	Required?
type?	Specifies the type of test to run (e.g. boot, migration etc)	yes
skip?	If equals 'yes', the test will not be executed	no
name?	The full **name** (not type) of the test (e.g. qcow2.ide.Fedora.8.32.install); see test configuration file format?. **This parameter is generated by the parser.**	yes
shortname?	The short **name** of the test (e.g. Fedora.8.32.install); see test configuration file format?. **This parameter is generated by the parser.** It specifies the tag to append to the Autotest test name, so that eventually the test name becomes something like kvm_runtest_2.Fedora.8.32.install.	yes
depend?	The full names of the dependencies of this test (e.g. ['qcow2.openSUSE-11.install', 'openSUSE-11.boot']). **This parameter is a list of strings, not a string, and is generated by the parser.** The test dispatcher will not run a test if one or more of its dependencies have run and failed.	yes

Parameters used by the preprocessor

The preprocessor runs before the test itself. It prepares VMs and images for the test, according to the supplied parameters.

Parameter	Effect/meaning	Required?
vms?	Lists the VM objects to be used in the test. Listed VMs that do not exist will be created. **Existing VMs that are not listed will be destroyed and removed.**	yes

VM preprocessor parameters

These parameters should be specified for each VM as explained above in addressing objects.

Parameter	Effect/meaning	Required?
start_vm?	If equals 'yes', the VM will be started **if it is down**; this parameter should be set to 'yes', for a certain VM object, in all tests that require the VM to be up.	no
restart_vm?	If equals 'yes', the VM will be (re)started, regardless of whether it's already up or not	no
start_vm_for_migrations?	If equals 'yes', the VM will be (re)started with the -incoming option so that it accepts incoming migrations; this parameter should be set to 'yes' for the destination VM object in a migration test.	no

The following parameters are remembered by a VM object when it is created or started. They cannot be changed while a VM is up. In order to change them, the VM must be restarted with new parameters.

Parameter	Effect/meaning	Required (when creating or starting a VM)
cdrom?	Specifies the name of an image file to be passed to QEMU with the -cdrom option. This is typically an ISO image file.	no
md5sum?	If specified, the VM will not be started (and thus the test will fail) if the MD5 sum of the **cdrom** image doesn't match this parameter. This is intended to verify the identity of the image used.	no
md5sum_1m?	Similar to **md5sum**, but specifies the MD5 sum of only the first MB of the **cdrom** image. If specified, this parameter is used instead of **md5sum**. Calculating the MD5 sum of the first MB of an image is much quicker than calculating it for the entire image.	no
mem	Specifies the amount of memory, in MB, the VM should have	yes
display	Selects the rendering method to be used by the VM; valid values are 'vnc', 'sdl' and 'nographic'. If 'vnc' is selected, the VM will be assigned an available VNC port automatically.	no
extra_params?	Specifies a string to append to the QEMU command line, e.g. '-snapshot'	no
use_telnet?	If equals 'yes', communication with the guest will be done via Telnet; otherwise SSH will be used.	no
ssh_port?	Specifies the guest's SSH/Telnet port; should normally be 22, unless Telnet is used, in which case this parameter should be 23.	if the VM should support SSH/Telnet communication
ssh_prompt?	A regular expression describing the guest's shell prompt	if the VM should support SSH/Telnet communication
username?	Specifies the username with which to attempt to log into the guest whenever necessary	if the VM should support SSH/Telnet communication
password?	Specifies the password with which to attempt to log into the guest whenever necessary	if the VM should support SSH/Telnet communication
cmd_shutdown?	Specifies the shell command to be used to shut the guest down (via SSH/Telnet) whenever necessary	if the VM should support being shutdown via SSH/Telnet
cmd_reboot?	Specifies the shell command to be used to reboot the guest (via SSH/Telnet) whenever necessary	if the VM should support rebooting via SSH/Telnet
images	Lists the image objects to be used by the VM	yes
nics	Lists the NIC objects to be used by the VM	yes

A VM will be restarted automatically if a parameter change leads to a different QEMU command line (for example, when `mem` changes). However, when other parameters change (such as **cmd_shutdown**) the VM will not be automatically restarted (unless **restart_vm** is set to 'yes'), and the change will have no effect.

Image preprocessor parameters

The following parameters should be specified for each image of each VM, as explained in addressing objects.

Parameter	Effect/meaning	Required?
create_image	If equals 'yes', the image file will be created using qemu-img **if it doesn't already exist**	no
force_create_image	If equals 'yes', the image file will be created using qemu-img regardless of whether it already exists. If the file already exists it will be overwritten by a blank image file.	no
image_name	Specifies the image filename **without the extension**	yes
image_format	Specifies the format of the image to be created/used, e.g. qcow2, raw, vmdk etc	yes
image_size	Specifies the size of the image to be created, in a format understood by qemu-img (e.g. 10G)	only when creating an image
drive_format	Specifies a string to pass to QEMU as the drive's 'if' parameter (e.g. ide, scsi)	no
image_snapshot?	If equals 'yes', 'snapshot=on' will be appended to the 'drive' option passed to QEMU	no
image_boot?	If equals 'yes', 'boot=on' will be appended to the 'drive' option passed to QEMU	no

NIC preprocessor parameters

The following parameters should be specified for each NIC of each VM, as explained in the section "addressing objects".

Parameter	Effect/meaning	Required?
nic_model?	A string to pass to QEMU as the NIC's 'model' parameter (e.g. e1000, virtio)	no

Parameters used by the postprocessor

The postprocessor runs after the test itself. It can shut down VMs, remove image files and clean up the test's results dir.

The suffix **_on_error** may be added to all parameters in this section (including VM and image parameters) to define special behavior for tests that fail or result in an error. The suffix should be added **after** all object addressing suffixes. If a parameter is specified without the suffix, it applies both when the test passes and when it fails. If a parameter is specified with the suffix, it applies only when the test fails, and overrides the parameter without the suffix.

For example, if we wanted the postprocessor to shut down **vm1** after the test, but only if the test failed, we'd write:

```
kill_vm_vm1_on_error = yes
```

If we wanted to shut down **another_vm** only if the test **passed**, we'd write:

```
kill_vm_another_vm = yes
kill_vm_another_vm_on_error = no
```

Since PPM files are normally used for debugging test failures, it would be very reasonable to choose to keep them only if the test fails. In that case we'd write:

```
keep_ppm_files = no
keep_ppm_files_on_error = yes
```

The following parameters define the postprocessor's behavior:

Parameter	Effect/meaning	Required?
vms?	Lists the VM objects to be handled by the postprocessor	yes
keep_ppm_files	If equals 'yes', the PPM image files in the test's debug directory will not be removed	no

VM postprocessor parameters

These parameters should be specified for each VM as explained above in "addressing objects".

Parameter	Effect/meaning	Required?
kill_vm	If equals 'yes', the VM will be shut down after the test	no
kill_vm_gracefully	If equals 'yes', and **kill_vm** equals 'yes', the first attempt to kill the VM will be done via SSH/Telnet with a clean shutdown command (rather than a quick 'quit' monitor command)	no
kill_vm_timeout	If **kill_vm** equals 'yes', this parameter specifies the time duration (in seconds) to wait for the VM to shut itself down, before attempting to shut it down externally; if this parameter isn't specified the VM killing procedure will start immediately following the test. This parameter is useful for tests that instruct a VM to shut down internally and need the postprocessor to shut it down only if it fails to shut itself down in a given amount of time	no
images	Lists the images objects, for this VM, to be handled by the postprocessor	no

Image postprocessor parameters

These parameters should be specified for each image of each VM as explained above in "addressing objects".

Parameter	Effect/meaning	Required?
remove_image	If equals 'yes', the image file will be removed after the test	no

Test parameters

Any number of additional parameters may be specified for each test, and they will be available for the test to use. See the tests? page for a list of tests and the parameters they use.

Real world example

The following example dictionary is taken from a dictionary list used in actual tests. The list was generated by the config file parser.

```
Dictionary #363:
    cmd_reboot = shutdown -r now
    cmd_shutdown = shutdown -h now
    depend = ['custom.qcow2.ide.default.up.Linux.Fedora.9.32.e1000.install',
         'custom.qcow2.ide.default.up.Linux.Fedora.9.32.e1000.setup',
         'custom.qcow2.ide.default.up.Linux.Fedora.9.32.default_nic.install',
         'custom.qcow2.ide.default.up.Linux.Fedora.9.32.default_nic.setup']
    drive_format = ide
    image_boot = yes
    image_format = qcow2
    image_name = fc9-32
    image_size = 10G
    images = image1
    keep_ppm_files = no
    keep_ppm_files_on_error = yes
    kill_vm = no
    kill_vm_gracefully = yes
    kill_vm_on_error = yes
    main_vm = vm1
    mem = 512
    migration_dst = dst
    migration_src = vm1
    migration_test_command = help
    name = custom.qcow2.ide.default.up.Linux.Fedora.9.32.e1000.migrate.1
    nic_model = e1000
    nics = nic1
    password = 123456
    shortname = Fedora.9.32.e1000.migrate.1
    ssh_port = 22
    ssh_prompt = \[root@.{0,50}][\#\$]
    start_vm = yes
    start_vm_for_migration_dst = yes
    type = migration
    username = root
    vms = vm1 dst
```

The test dispatching system

This test's **name** is a rather long string that indicates all the variants this test belongs to; its **shortname**, however, is much shorter: **Fedora.9.32.e1000.migrate.1**.

The test depends on 4 other tests, as indicated by the depend? parameter. The listed strings are the **names** of these tests. If any of these 4 tests runs and fails, the current test will be skipped.

Preprocessing

This test requires two VMs as indicated by the vms? parameter: one will be called **vm1** and the other **dst**. The parameter **start_vm**, which lacks a VM suffix and therefore applies to both VMs, indicates that if any of these VM objects does not exist or is not up, it will be started. However, **start_vm_for_migration_dst = yes** indicates that the VM **dst** should be started with the -incoming option so that it accepts an incoming migration.

The images parameter indicates that a single image object will be used by each VM, and they will both be called **image1**. This poses no problem because an image object only exists within the scope of its owner VM. However, both image objects actually point to the same image file, as indicated by **image_name = fc9-32**. If image_name appeared with some suffix (e.g. **image_name_image1_vm1** or **image_name_vm1**) it would be attributed to a single VM, not

both. **image_format = qcow2** indicates that this is a qcow2 image file, so the actual filename becomes fc9-32.qcow2. **image_boot = yes** instructs the preprocessor to add ',boot=on' to the -drive option in the QEMU command line. **drive_format = ide** adds ',if=ide'. No image file is created during the preprocessing phase of this test because both **create_image** and **force_create_image** are not specified.

The **nics** parameter indicates that each VM should be equipped with a single NIC object named **nic1**. **nic_model = e1000** indicates that all NICs (due to the lack of a suffix) should be of the e1000 model. If one wished to specify a different NIC model for each VM, one could specify, for example, **nic_model_vm1 = e1000** and **nic_model_dst = rtl8139**.

The parameters mem, ssh_port?, ssh_prompt?, username?, password?, cmd_reboot? and cmd_shutdown? apply to both VMs. See #VM preprocessor parameters for an explanation of these parameters.

The test itself

The parameters migration_src?, migration_dst? and migration_test_command? are used by the migration test. They instruct it to migrate from **vm1** to **dst** and use the shell command **help** to test that the VM is alive following the migration.

The parameter **main_vm** happens to be specified because the format of the configuration file makes it easy to set a parameter for a large group of tests. However, in the case of a migration test, this parameter is not used and its presence is harmless.

Postprocessing

keep_ppm_files = no and keep_ppm_files_on_error = yes? indicate that normally the PPM files (images left in the test's 'debug' directory) will not be kept; however, if the test fails, they will. This makes sense because the PPM files take quite a lot of hard drive space, and they are mostly useful to debug failures.

kill_vm = no indicates that normally both VMs should be left alone following the test.

kill_vm_on_error = yes? indicates that in the case of a failure, both VMs should be destroyed. This makes sense because if a migration test fails, the VMs involved may not be functional for the next test, thus causing it to fail.

If they are killed by the postprocessor, the preprocessor of the next test will automatically start them, assuming start_vm = yes? is specified for the next test. The parameter kill_vm_gracefully indicates that if a VM is to be killed, it should first be attempted via SSH/Telnet with a shutdown shell command, specified by the cmd_shutdown? parameter.

8.1.2 Cartesian Config Reference

bridge

Description

Sets the name of the bridge to which a VM nic will be added to. This only applies to scenarios where 'nic_mode' is set to 'tap'.

It can be set as a default to all nics:

```
bridge = virbr0
```

Or to a specific nic, by prefixing the parameter key with the nic name, that is for attaching 'nic1' to bridge 'virbr1':

```
bridge_nic1 = virbr1
```

Defined On

- shared/cfg/guest-hw.cfg
- shared/cfg/machines.cfg

Used By

- virttest/qemu_vm.py

Referenced By

No other documentation currently references this configuration key.

cdroms

Description

Sets the list of cdrom devices that a VM will have.

Usually a VM will start with a single cdrom, named 'cd1'.

```
cdroms = cd1
```

But a VM can have other cdroms such as 'unattended' for unattended installs:

```
variants:
    - @Linux:
        unattended_install:
            cdroms += " unattended"
```

And 'winutils' for Microsoft Windows VMs:

```
variants:
    - @Windows:
        unattended_install.cdrom, whql.support_vm_install:
            cdroms += " winutils"
```

Defined On

- shared/cfg/base.cfg
- shared/cfg/virtio-win.cfg

Used By

- virttest/qemu_vm.py

Referenced By

No other documentation currently references this configuration key.

cd_format

Description

Sets the format for a given cdrom drive. This directive exists to do some special magic for cd drive formats 'ahci' and 'usb2' (see virttest/qemu_vm.py for more information).

Currently used options in Avocado-VT are: ahci and usb2.

Example:

```
variants:
    - usb.cdrom:
        cd_format = usb2
```

Defined On

- shared/cfg/virtio-win.cfg
- shared/cfg/guest-hw.cfg

Used By

- virttest/qemu_vm.py

Referenced By

No other documentation currently references this configuration key.

See also

- drive_format

check_image

Description

Configures if we want to run a check on the image files during post processing. A check usually means running 'qemu-img info' and 'qemu-img check'.

This is currently only enabled when image_format is set to 'qcow2'.

```
variants:
    - @qcow2:
        image_format = qcow2
        check_image = yes
```

Defined On

- shared/cfg/guest-hw.cfg

Used By

- virttest/env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- images
- image_name
- image_format
- create_image
- remove_image

convert_ppm_files_to_png

Description

Configures whether files generated from screenshots in PPM format should be automatically converted to PNG files.

```
convert_ppm_files_to_png = yes
```

Usually we're only interested in spending time converting files for easier viewing on situations with failures:

```
convert_ppm_files_to_png_on_error = yes
```

Defined On

The stock configuration key (without suffix) is not currently defined on any sample cartesian configuration file.

The configuration key with the 'on_error' suffix is defined on:

- shared/cfg/base.cfg

Used By

- virttest/env_process.py

Referenced By

No other documentation currently references this configuration key.

create_image

Description

Configures if we want to create an image file during pre processing, if it does **not** already exists. To force the creation of the image file even if it already exists, use force_create_image.

To create an image file if it does **not** already exists:

```
create_image = yes
```

Defined On

- shared/cfg/guest-hw.cfg

Used By

- virttest/env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- images
- image_name
- image_format
- create_image
- force_create_image
- remove_image

display

Description

Sets the VM display type. Of course, only one display type is allowed, and current valid options are: vnc, sdl, spice and nographic.

```
display = vnc
```

For VNC displays, the port number is dynamically allocated within the 5900 - 6100 range.

```
display = sdl
```

An SDL display does not use a port, but simply behaves as an X client. If you want to send the SDL display to a different X Server, see x11_display?

```
display = spice
```

For spice displays, the port number is dynamically allocated within the 8000 - 8100 range.

```
display = nographic
```

nographic for qemu/kvm means that the VM will have no graphical display and that serial I/Os will be redirected to console.

Defined On

- shared/cfg/base.cfg

Used By

- virttest/qemu_vm.py

Referenced By

No other documentation currently references this configuration key.

drive_cache

Description

Sets the caching mode a given drive. Currently the valid values are: writethrough, writeback, none and unsafe.

Example:

```
drive_cache = writeback
```

This option can also be set specifically to a drive:

```
drive_cache_cd1 = none
```

Defined On

- shared/cfg/base.cfg

Used By

- virttest/qemu_vm.py

Referenced By

No other documentation currently references this configuration key.

drive_format

Description

Sets the format for a given drive.

Usually this passed directly to qemu 'if' sub-option of '-drive' command line option. But in some special cases, such as when drive_format is set to 'ahci' or 'usb2', some special magic happens (see client/virt/kvm_vm.py for more information).

Currently available options in qemu include: ide, scsi, sd, mtd, floppy, pflash, virtio.

Currently used options in Avocado-VT are: ide, scsi, virtio, ahci, usb2.

Example:

```
drive_format = ide
```

Defined On

- shared/cfg/guest-hw.cfg

Used By

- virttest/qemu_vm.py

Referenced By

No other documentation currently references this configuration key.

drive_index

Description

Sets the index, that is, ordering precedence of a given drive. Valid values are integers starting with 0.

Example:

```
drive_index_image1 = 0
drive_index_cd1 = 1
```

This will make the drive that has 'image1' appear before the drive that has 'cd1'.

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/guest-os.cfg.sample
- client/tests/kvm/subtests.cfg.sample
- client/tests/kvm/virtio-win.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

drive_werror

Description

Sets the behavior for the VM when a drive encounters a read or write error. This is passed to QEMU 'werror' sub-option of the '-drive' command line option.

Valid for QEMU are: ignore, stop, report, enospc.

Example:

```
drive_werror = stop
```

Defined On

- client/tests/kvm/subtests.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

drive_serial

Description

Sets the serial number to assign to the drive device.

Defined On

This configuration key is not currently defined on any sample cartesian configuration file.

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

file_transfer_client

Description

Sets the kind of application, thus protocol, that will be spoken when transfering files to and from the guest.

Avocado-VT currently allows for two options: 'scp' or 'rss'.

For Linux VMs, we default to SSH:

```
variants:
    - @Linux:
        file_transfer_client = scp
```

And for Microsoft Windows VMs we default to rss:

```
variants:
    - @Windows:
        file_transfer_client = rss
```

Defined On

- client/tests/kvm/guest-os.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

See Also

- redirs
- file_transfer_port
- guest_port_file_transfer

file_transfer_port

Description

Sets the port on which the application used to transfer files to and from the guest will be listening on.

When file_transfer_client is scp, this is by default 22:

```
variants:
    - @Linux:
        file_transfer_client = scp
        file_transfer_port = 22
```

And for rss, the default is port 10023:

```
variants:
    - @Windows:
        file_transfer_client = rss
        file_transfer_port = 10023:
```

Defined On

- client/tests/kvm/guest-os.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

See Also

- redirs
- file_transfer_client
- guest_port_file_transfer

force_create_image

Description

Configures if we want to create an image file during pre processing, **even if it already exists**. To create an image file only if it **does not** exist, use create_image instead.

To create an image file **even if it already exists**:

```
force_create_image = yes
```

Defined On

- client/tests/kvm/subtests.cfg.sample

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- images
- image_name
- image_format
- create_image
- check_image
- remove_image

guest_port

Description

guest_port is not a configuration item itself, but the basis (prefix) of other real configuration items such as:

- guest_port_remote_shell
- guest_port_file_transfer
- guest_port_unattended_install

Defined On

Variations of guest_port are defined on the following files:

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/guest-os.cfg.sample
- client/tests/kvm/subtests.cfg.sample

Referenced By

No other documentation currently references this configuration key.

See also

- redirs

guest_port_remote_shell

Description

Sets the port of the remote shell server that runs inside guests. On Linux VMs, this is the set by default to the standard SSH port (22), and for Windows guests, set by default to port 10022.

This is a specialization of the guest_port configuration entry.

Example, default entry:

```
guest_port_remote_shell = 22
```

Overridden on Windows variants:

```
variants:
    - @Windows:
        guest_port_remote_shell = 10022
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/guest-os.cfg.sample

Used By

- client/virt/kvm_vm.py
- client/virt/virt_utils.py

Referenced By

No other documentation currently references this configuration key.

See Also

- redirs
- shell_port?

guest_port_file_tranfer

Description

Sets the port of the server application running inside guests that will be used for transferring files to and from this guest.

On Linux VMs, the file_transfer_client is set by default to 'scp', and this the port is set by default to the standard SSH port (22).

For Windows guests, the file_transfer_client is set by default to 'rss', and the port is set by default to 10023.

This is a specialization of the guest_port configuration entry.

Example, default entry:

```
guest_port_file_transfer = 22
```

Overridden on Windows variants:

```
variants:
    - @Windows:
        guest_port_file_transfer = 10023
```

Defined On

- client/tests/kvm/guest-os.cfg.sample

Used By

- client/virt/kvm_vm.py
- client/virt/virt_utils.py

Referenced By

No other documentation currently references this configuration key.

See Also

- redirs
- file_transfer_port
- file_transfer_client

guest_port_unattended_install

Description

Sets the port of the helper application/script running inside guests that will be used for flagging the end of the unattended install.

Both on Linux and Windows VMs, the default value is 12323:

```
guest_port_unattended_install = 12323
```

This must match with the port number on unattended install files. On Linux VMs, this is hardcoded on kickstart files '%post' section:

```
%post --interpreter /usr/bin/python
...
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('', 12323))
server.listen(1)
(client, addr) = server.accept()
client.send("done")
client.close()
```

This is a specialization of the guest_port configuration entry.

Defined On

- client/tests/kvm/subtests.cfg.sample

Used By

- client/tests/kvm/tests/unattended_install.py

Referenced By

No other documentation currently references this configuration key.

See Also

- redirs

images

Description

Sets the list of disk devices (backed by a image file or device) that a VM will have.

Usually a VM will start with a single image, named image1:

```
images = image1
```

But a VM can have other images. One example is when we test the maximum number of disk devices supported on a VM:

```
# Tests
variants:
    - multi_disk: install setup image_copy unattended_install.cdrom
        variants:
            - max_disk:
                images += " stg stg2 stg3 stg4 stg5 stg6 stg7 stg8 stg9 stg10 stg11␣
↪stg12 stg13 stg14 stg15 stg16 stg17 stg18 stg19 stg20 stg21 stg22 stg23"
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/guest-os.cfg.sample
- client/tests/kvm/subtests.cfg.sample

Used By

- client/virt/kvm_vm.py
- virt_env_process.py
- client/tests/kvm/tests/enospc.py
- client/tests/kvm/tests/image_copy.py

Referenced By

No other documentation currently references this configuration key.

images_good

Description

Sets the URI of a NFS server that hosts "good" (think "golden") images, that will be copied to the local system prior to running other tests.

The act of copying of "good" images is an alternative to installing a VM from scratch before running other tests.

The default value is actually an invalid value that must be changed if you intend to use this feature:

```
images_good = 0.0.0.0:/autotest/images_good
```

Defined On

- client/tests/kvm/base.cfg.sample

Used By

- client/virt/tests/image_copy.py

Referenced By

No other documentation currently references this configuration key.

image_format

Description

Sets the format of the backing image file for a given drive.

The value of this configuration key is usually passed verbatim to image creation commands. It's worth noticing that QEMU has support for many formats, while Avocado-VT currently plays really well only with **qcow2** and **raw**.

You can also use **vmdk**, but it's considered 'not supported', at least on image conversion tests.

To set the default image format:

```
image_format = qcow2
```

To set the image format for another image:

```
# Tests
variants:
    - block_hotplug: install setup image_copy unattended_install.cdrom
        images += " stg"
        image_format_stg = raw
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/subtests.cfg.sample

Used By

- client/virt/virt_vm.py
- client/tests/kvm/tests/qemu_img.py

Referenced By

No other documentation currently references this configuration key.

See Also

- images
- image_name
- image_size

image_name

Description

Sets the name of an image file.

If the image file is not a block device (see image_raw_device) the actual file created will be named accordingly (together with the extension, according to image_format).

When this configuration key is used without a suffix, it's setting the name of all images without a specific name. The net effect is that it sets the name of the 'default' image. Example:

```
# Guests
variants:
    - @Linux:
        variants:
            - Fedora:
                variants:
                    - 15.64:
                        image_name = f15-64
```

This example means that when a Fedora 15 64 bits is installed, and has a backing image file created, it's going to be named starting with 'f15-64'. If the image_format specified is 'qcow2', then the complete filename will be 'f15-64.qcow2'.

When this configuration key is used with a suffix, it sets the name of a specific image. Example:

```
# Tests
variants:
    - block_hotplug: install setup image_copy unattended_install.cdrom
        images += " stg"
        image_name_stg = storage
```

Defined On

- client/tests/kvm/guest-os.cfg.sample
- client/tests/kvm/subtests.cfg.sample
- client/tests/kvm/tests.cfg.sample

Used By

- client/virt/kvm_vm.py
- client/tests/kvm/tests/qemu_img.py

Referenced By

- How to run virt-test tests on an existing guest image?

See Also

- images
- image_format
- image_raw_device

image_raw_device

Description

Flags whether the backing image for a given drive is a block device instead of a regular file.

By default we assume all images are backed by files:

```
image_raw_device = no
```

But suppose you define a new variant, for another guest, that will have a disk backed by a block device (say, an LVM volume):

```
CustomGuestLinux:
    image_name = /dev/vg/linux_guest
    image_raw_device = yes
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/tests.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

- How to run virt-test tests on an existing guest image?

image_size

Description

Sets the size of image files. This applies to images creation and also validation tests (when checking that a image was properly created according to what was requested).

By default the image size is set to 10G:

```
image_size = 10G
```

But a VM can have other drives, backed by other image files (or block devices), with different sizes:

```
# Tests
variants:
    - block_hotplug: install setup image_copy unattended_install.cdrom
        images += " stg"
        boot_drive_stg = no
        image_name_stg = storage
        image_size_stg = 1G
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/guest-os.cfg.sample
- client/tests/kvm/subtests.cfg.sample
- client/tests/kvm/tests.cfg.sample

Used By

- client/virt/kvm_vm.py
- client/tests/kvm/tests/qemu_img.py

Referenced By

No other documentation currently references this configuration key.

See Also

- images
- image_name
- image_format

keep_ppm_files

Description

Configures whether should we keep the original screedump files in PPM format when converting them to PNG, according to convert_ppm_files_to_png

To keep the PPM files:

```
keep_ppm_files = yes
```

To keep the PPM files only on situations with failures:

```
keep_ppm_files_on_error = yes
```

Defined On

This configuration key is not currently defined on any sample cartesian configuration file, but a sample (commented out) appears on:

- client/tests/kvm/base.cfg.sample

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

keep_screendumps

Description

Flags whether screendumps (screenshots of the VM console) should be kept or delete during post processing.

To keep the screendumps:

```
keep_screendumps = yes
```

Usually we're only interested in keeping screendumps on situations with failures, to ease the debugging:

```
keep_screendumps_on_error = yes
```

Defined On

The stock configuration key (without suffix) is not currently defined on any sample cartesian configuration file.

The configuration key with the 'on_error' suffix is defined on:

- client/tests/kvm/base.cfg.sample

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

kill_unresponsive_vms

Description

Configures whether VMs that are running, but do not have a responsive session (for example via SSH), should be destroyed (of course, not gracefully) during post processing.

This behavior is enabled by default. To turn it off and leave unresponsive VMs lying around (usually **not** recommended):

```
kill_unresponsive_vms = no
```

Defined On

- client/tests/kvm/base.cfg.sample

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See also

- kill_vm
- kill_vm_timeout
- kill_vm_gracefully

kill_vm

Description

Configures whether a VM should be shutdown during post processing. How exactly the VM will be shutdown is configured by other parameters such as kill_vm_gracefully and kill_vm_timeout.

To force shutdown during post processing:

```
kill_vm = yes
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/guest-os.cfg.sample
- client/tests/kvm/subtests.cfg.sample

- client/tests/kvm/unittests.cfg.sample

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See also

- kill_vm_timeout
- kill_vm_gracefully
- kill_unresponsive_vms

kill_vm_gracefully

Description

Flags whether a graceful shutdown command should be sent to the VM guest OS before attempting to either halt the VM at the hypervisor side (sending an appropriate command to QEMU or even killing its process).

Of course, this is only valid when kill_vm is set to 'yes'.

To force killing VMs without using a graceful shutdown command (such as 'shutdown -h now'):

```
kill_vm_gracefully = no
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/guest-os.cfg.sample
- client/tests/kvm/subtests.cfg.sample
- client/tests/kvm/unittests.cfg.sample

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See also

- kill_vm
- kill_vm_timeout
- kill_unresponsive_vms

kill_vm_timeout

Description

Configures the amount of time, in seconds, to wait for VM shutdown during the post processing.

This is only relevant if kill_vm is actually set to 'yes'.

To set the timeout to one minute:

```
kill_vm_timeout = 60
```

Defined On

- client/tests/kvm/guest-os.cfg.sample
- client/tests/kvm/subtests.cfg.sample

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See also

- kill_vm
- kill_vm_gracefully
- kill_unresponsive_vms

login_timeout

Description

Sets the amount of time, in seconds, to wait for a session (SSH/Telnet/Netcat) with the VM.

To set the timeout to 6 minutes:

```
login_timeout = 360
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/subtests.cfg.sample

Used By

- client/virt/tests/autotest.py
- client/virt/tests/boot.py
- client/virt/tests/clock_getres.py
- client/virt/tests/ethtool.py
- client/virt/tests/file_transfer.py
- client/virt/tests/fillup_disk.py
- client/virt/tests/guest_s4.py
- client/virt/tests/guest_test.py
- client/virt/tests/iofuzz.py
- client/virt/tests/ioquit.py
- client/virt/tests/iozone_windows.py
- client/virt/tests/jumbo.py
- client/virt/tests/kdump.py
- client/virt/tests/linux_s3.py
- client/virt/tests/lvm.py
- client/virt/tests/mac_change.py
- client/virt/tests/multicast.py
- client/virt/tests/netperf.py
- client/virt/tests/nicdriver_unload.py
- client/virt/tests/nic_promisc.py
- client/virt/tests/ping.py
- client/virt/tests/shutdown.py
- client/virt/tests/softlockup.py
- client/virt/tests/stress_boot.py
- client/virt/tests/vlan.py
- client/virt/tests/watchdog.py
- client/virt/tests/whql_client_install.py
- client/virt/tests/whql_submission.py

- client/virt/tests/yum_update.py
- client/tests/kvm/tests/balloon_check.py
- client/tests/kvm/tests/cdrom.py
- client/tests/kvm/tests/cpu_hotplug.py
- client/tests/kvm/tests/enospc.py
- client/tests/kvm/tests/floppy.py
- client/tests/kvm/tests/hdparm.py
- client/tests/kvm/tests/migration_multi_host.py
- client/tests/kvm/tests/migration.py
- client/tests/kvm/tests/migration_with_file_transfer.py
- client/tests/kvm/tests/migration_with_reboot.py
- client/tests/kvm/tests/multi_disk.py
- client/tests/kvm/tests/nic_bonding.py
- client/tests/kvm/tests/nic_hotplug.py
- client/tests/kvm/tests/nmi_watchdog.py
- client/tests/kvm/tests/pci_hotplug.py
- client/tests/kvm/tests/physical_resources_check.py
- client/tests/kvm/tests/qemu_img.py
- client/tests/kvm/tests/set_link.py
- client/tests/kvm/tests/smbios_table.py
- client/tests/kvm/tests/stop_continue.py
- client/tests/kvm/tests/system_reset_bootable.py
- client/tests/kvm/tests/timedrift.py
- client/tests/kvm/tests/timedrift_with_migration.py
- client/tests/kvm/tests/timedrift_with_reboot.py
- client/tests/kvm/tests/timedrift_with_stop.py
- client/tests/kvm/tests/trans_hugepage_defrag.py
- client/tests/kvm/tests/trans_hugepage.py
- client/tests/kvm/tests/trans_hugepage_swapping.py
- client/tests/kvm/tests/usb.py
- client/tests/kvm/tests/vmstop.py

Referenced By

No other documentation currently references this configuration key.

main_monitor

Description

Sets the default monitor for a VM, meaning that when a test accesses the **monitor** property of a **VM** class instance, that one monitor will be returned.

Usually a VM will have a single monitor, and that will be a regular Human monitor:

```
main_monitor = humanmonitor1
```

If a **main_monitor** is not defined, the **monitor** property of a **VM** class instance will assume that the first monitor set in the monitors list is the main monitor.

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/unittests.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

See Also

- monitors
- monitor_type
- client/virt/kvm_monitor.py

main_vm

Description

Sets name of the main VM.

There's nothing special about this configuration item, except that most tests will also reference its value when fetching a VM from the Environment (see class **Env** on file client/virt/virt_utils.py).

The default name of the main VM is **vm1**:

```
main_vm = vm1
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/unittests.cfg.sample

Referenced By

No other documentation currently references this configuration key.

mem

Description

Sets the amount of memory (in MB) a VM will have.

The amount of memory a VM will have for most tests is of the main VM is 1024:

```
mem = 1024
```

But for running KVM unittests, we currently set that to 512:

```
mem = 512
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/unittests.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

migration_mode

Description

If migration mode is specified, the VM will be started in incoming mode for migration. Valid modes for migration are: **tcp**, **unix** and **exec**.

To start a VM in incoming mode for receiving migration data via tcp:

```
migration_mode = tcp
```

A port will be allocated from the range 5200 to 5899.

Defined On

This configuration item is currently not defined on a sample cartesian configuration file.

Used By

- client/tests/kvm/migration_control.srv

Referenced By

No other documentation currently references this configuration key.

monitors

Description

Sets the list of monitors that a VM currently has running. See [QEMU has two types of monitors:

- The regular, also known as Human monitor, intended for interaction with people (but also very much used by other tools, Autotest inclusive)
- The QMP monitor, a monitor that speaks the QMP protocol.

Usually a VM will have a single monitor, and that will be a regular Human monitor:

```
monitors = humanmonitor1
main_monitor = humanmonitor1
monitor_type_humanmonitor1 = human
monitor_type = human
```

The monitor type is defined by monitor_type.

Here's a more detailed exaplanation of the configuration snippet above:

```
monitors = humanmonitor1
```

The default VM will have only one monitor, named **humanmonitor1**.

```
main_monitor = humanmonitor1
```

The main monitor will also be **humanmonitor1**. When a test has to talk to a monitor, it usually does so through the main monitor.

```
monitor_type_humanmonitor1 = human
```

This configuration sets the specific type of the **humanmonitor1** to be **human**.

```
monitor_type = human
```

And finally this configuration sets the default monitor type also to be **human**.

Suppose you define a new monitor for your VMs:

```
monitors += ' monitor2'
```

Unless you also define:

```
monitor_type_monitor2 = qmp
```

monitor2 will also be a human monitor.

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/unittests.cfg.sample

Used By

- client/tests/kvm/kvm.py
- client/virt/kvm_vm.py
- client/virt/virt_test_utils.py

Note: most tests that interact with the monitor do so through the **monitor** property of the **VM** class, and not by evaluating this parameter value. This is usally only done by the **VM** class.

Referenced By

No other documentation currently references this configuration key.

See Also

- client/virt/kvm_monitor.py

monitor_type

Description

Sets the type of the monitor. QEMU has two types of monitors:

- The regular, also known as Human monitor, intended for interaction with people (but also very much used by other tools, Autotest inclusive)
- The QMP monitor, a monitor that speaks the QMP protocol.

To set the default monitor type to be a QMP monitor:

```
monitor_type = qmp
```

To set the type of a specific monitor use:

```
monitor_type_humanmonitor1 = human
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/unittests.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

See Also

- client/virt/kvm_monitor.py

nic_mode

Description

Configures the mode of a Network Interface Card.

Suitable values for this configuration item are either **user** or **tap**.

User mode networking is the default **on QEMU**, but Tap mode is the current default in Autotest:

```
nic_mode = tap
```

When **nic_mode** is set to Tap you should also set a bridge.

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/subtests.cfg.sample
- client/tests/kvm/migration_control.srv

Used By

- client/virt/kvm_vm.py
- client/tests/kvm/tests/physical_resources_check.py

Referenced By

No other documentation currently references this configuration key.

See Also

- bridge
- redirs

nics

Description

Sets the list of network interface cards that a VM will have.

Usually a VM will start with a single nic, named nic1:

```
nics = nic1
```

But a VM can have other nics. Some tests (usually network related) add other nics. One obvious example is the bonding test:

```
# Tests
variants:
    - nic_bonding: install setup image_copy unattended_install.cdrom
        nics += ' nic2 nic3 nic4'
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/subtests.cfg.sample

Used By

- client/virt/virt_vm.py
- client/virt/kvm_vm.py
- client/tests/kvm/tests/nic_bonding.py
- client/tests/kvm/tests/physical_resources_check.py

Referenced By

No other documentation currently references this configuration key.

pre_command

Description

Configures a command to be executed during pre processing.

The pre processing code will execute the given command, waiting for an amount of time and failing the test unless the command is considered noncritical.

Defined On

This configuration key is not currently defined on any sample cartesian configuration file in its stock format.

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- pre_command_timeout
- pre_command_non_critical?

pre_command_timeout

Description

Configures the amount of time to wait while executing a command during pre processing.

Defined On

This configuration key is not currently defined on any sample cartesian configuration file in its stock format.

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- pre_command
- pre_command_non_critical?

pre_command_noncritical

Description

Flags if the command configured to to be executed during pre processing, is not critical, that is, if an error during its execution should only logged or fail the test.

Defined On

This configuration key is not currently defined on any sample cartesian configuration file in its stock format.

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- pre_command
- pre_command_timeout

profilers

Description

Sets the list of Autotest profilers to be enabled during the test run (they're removed from the job's list of profilers when the test finishes).

This is commonly used to enable the kvm_stat profiler:

```
profilers = kvm_stat
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/subtests.cfg.sample

Used By

- client/virt/virt_utils.py

Referenced By

No other documentation currently references this configuration key.

See Also

- Setting up profiling on virt-test
- Using and developing job profilers

post_command

Description

Configures a command to be executed during post processing.

The pre processing code will execute the given command, waiting for an amount of time and failing the test unless the command is considered noncritical.

Defined On

This configuration key is not currently defined on any sample cartesian configuration file in its stock format.

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- post_command_timeout
- post_command_non_critical?

post_command_timeout

Description

Configures the amount of time to wait while executing a command during post processing.

Defined On

This configuration key is not currently defined on any sample cartesian configuration file in its stock format.

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- post_command
- post_command_non_critical?

post_command_noncritical

Description

Flags if the command configured to to be executed during pre processing, is not critical, that is, if an error during its execution should only logged or fail the test.

Defined On

This configuration key is not currently defined on any sample cartesian configuration file in its stock format.

Used By

- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- post_command
- post_command_timeout

qemu_binary

Description

Sets either the name or full path for the QEMU binary.

By default this is as simple as possible:

```
qemu_binary = qemu
```

But while testing the qemu-kvm userspace, one could use:

```
qemu_binary = /usr/bin/qemu-kvm
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/tests.cfg.sample
- client/tests/kvm/unittests.cfg.sample

Used By

- client/virt/kvm_vm.py
- client/virt/virt_env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- qemu_img_binary

qemu_img_binary

Description

Sets either the name or full path for the **qemu-img** binary.

By default this is as simple as possible:

```
qemu_img_binary = qemu-img
```

Defined On

- client/tests/kvm/base.cfg.sample
- client/tests/kvm/tests.cfg.sample
- client/tests/kvm/unittests.cfg.sample

Used By

- client/virt/virt_vm.py
- client/tests/kvm/tests/qemu_img.py

Referenced By

No other documentation currently references this configuration key.

See Also

- qemu_binary

qxl_dev_nr

Description

Sets the number of display devices available through SPICE. This is only valid when qxl is set.

The default configuration enables a single display device:

```
qxl_dev_nr = 1
```

Note that due to a limitation in the current Autotest code (see client/virt/kvm_vm.py) this setting is only applied when the QEMU syntax is:

```
# qemu -qxl 2
```

and not applied when the syntax is:

```
# qemu -vga qxl
```

Defined On

- client/tests/kvm/base.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

See Also

- qxl
- vga?
- display

qxl

Description

Flags if the VGA device should be an of type **qxl**.

The default configuration enables a **qxl** VGA:

```
qxl = on
```

Note that if vga? is also set, **qxl** takes precedence over it.

Defined On

- client/tests/kvm/base.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

See Also

- qxl_dev_nr
- vga?

redirs

Description

Sets the network redirections between host and guest. These are only used and necessary when using 'user' mode network.

Example:

```
redirs = remote_shell
guest_port_remote_shell = 22
```

A port will be allocated on the host, usually within the range 5000-5899, and all traffic to/from this port will be redirect to guest's port 22.

Defined On

- shared/cfg/base.cfg

Used By

- virttest/qemu_vm.py

Referenced By

No other documentation currently references this configuration key.

See also

- guest_port
- guest_port_remote_shell

remove_image

Description

Configures if we want to remove image files during post processing.

To keep all images after running tests:

```
remove_image = no
```

On a test with multiple transient images, to remove all but the main image (**image1**), use:

```
remove_image = yes
remove_image_image1 = no
```

Defined On

- shared/cfg/base.cfg

Used By

- virttest/env_process.py

Referenced By

No other documentation currently references this configuration key.

See Also

- images
- image_name
- image_format
- create_image
- force_create_image

spice

Description

Sets extra arguments to be passed to the QEMU **-spice** command line argument.

Note that there's no need to pass a port number, as this will be automatically allocated from the 8000 - 8100 range.

By default, the extra arguments disable authentication:

```
spice = disable-ticketing
```

Defined On

- client/tests/kvm/base.cfg.sample

Used By

- client/virt/kvm_vm.py

Referenced By

No other documentation currently references this configuration key.

See Also

- qxl
- qxl_dev_nr
- vga?
- display

8.2 Indices and tables

- genindex
- modindex
- search

Python Module Index

V

Index

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W

X

Y