avocado-i2n Documentation

Intra2net AG

Jun 30, 2023

Contents

CHAPTER 1

avocado_i2n package

1.1 Subpackages

1.1.1 avocado_i2n.cartgraph package

1.1.1.1 Submodules

avocado_i2n.cartgraph.graph module

SUMMARY

Utility for the main test suite data structure.

Copyright: Intra2net AG

INTERFACE

avocado_i2n.cartgraph.graph.**set_graph_logging_level**(*level=20*)
Set the logging level specifically for the Cartesian graph.

This determines what descriptions of the graph will be dumped for debugging purposes.

class avocado_i2n.cartgraph.graph.**TestGraph**
Bases: object

The main parsed and traversed test data structure.

This data structure uses a tree for each test object all of which overlap in a directed graph. All tests are using objects that can be brought to certain states and need some specific setup. All states can thus be saved and reused for other tests, resulting in a tree structure of derived states for each object. These object trees are then interconnected as a test might use multiple objects (vms) at once resulting in a directed graph. Running all tests is nothing more but traversing this graph in DFS-like way to minimize setup repetition. The policy of this traversal determines whether an automated setup (tests not defined by the user but needed for his/her tests) will

be performed, ignored, overwritten, etc. The overall graph is extracted from the given Cartesian configuration, expanding Cartesian products of tests and tracing their object dependencies.

suffixes
Test object suffixes and their variant restrictions.

prefixes
Test node prefixes and their variant restrictions.

__init__()
Construct the test graph.

__repr__()
Return repr(self).

new_objects(*objects*)
Add new objects excluding (old) repeating ones as ID.

Parameters **objects** ([TestObject] or TestObject) – candidate test objects

new_nodes(*nodes*)
Add new nodes excluding (old) repeating ones as ID.

Parameters **nodes** ([TestNode] or TestNode) – candidate test nodes

load_setup_list(*dump_dir*, *filename='setup_list'*)
Load the setup state of each node from a list file.

Parameters

- **dump_dir** (*str*) – directory for the dump image
- **filename** (*str*) – file to load the setup information from

save_setup_list(*dump_dir*, *filename='setup_list'*)
Save the setup state of each node to a list file.

Parameters

- **dump_dir** (*str*) – directory for the dump image
- **filename** (*str*) – file to save the setup information to

report_progress()
Report the total test run progress as the number and percentage of tests that are fully finished (will not be run again).

The estimation includes setup tests which might be reused and therefore provides worst case scenario for the number of remaining tests. It also does not take into account the duration of each test which could vary significantly.

visualize(*dump_dir*, *tag=0*)
Dump a visual description of the Cartesian graph at a given parsing/traversal step.

Parameters

- **dump_dir** (*str*) – directory for the dump image
- **tag** (*str*) – tag of the dump, e.g. parsing/traversal step and slot

flag_children(*node_name=None*, *object_name=None*, *flag_type='run'*, *flag=True*, *skip_roots=False*)
Set the run/clean flag for all children of a parent node of a given name or the entire graph.

Parameters

- **node_name** (*str or None*) – name of the parent node or root if None

- **object_name** (*str or None*) – test object whose state is set or shared root if None
- **flag_type** (*str*) – 'run' or 'clean' categorization of the children
- **flag** (*bool*) – whether the run/clean action should be executed or not
- **skip_roots** (*bool*) – whether the roots should not be flagged as well

Raises AssertionError if obtained # of root tests is != 1

flag_parent_intersection(*graph*, *flag_type='run'*, *flag=True*, *skip_object_roots=False*, *skip_shared_root=False*)

Intersect the test nodes with the test nodes from another graph and set a run/clean flag for each one in the intersection.

Parameters

- **graph** (*TestGraph*) – Cartesian graph to intersect the current graph with
- **flag_type** (*str*) – 'run' or 'clean' categorization of the children
- **flag** (*bool*) – whether the run/clean action should be executed or not
- **skip_object_roots** (*bool*) – whether the object roots should not be flagged as well
- **skip_shared_root** (*bool*) – whether the shared root should not be flagged as well

avocado_i2n.cartgraph.node module

SUMMARY

Utility for the main test suite substructures like test nodes.

Copyright: Intra2net AG

INTERFACE

class avocado_i2n.cartgraph.node.**TestNode**(*prefix*, *config*, *object*)

Bases: object

A wrapper for all test relevant parts like parameters, parser, used objects and dependencies to/from other test nodes (setup/cleanup).

params
Parameters (cache) property.

final_restr
Final restriction to make the object parsing variant unique.

long_prefix
Sufficiently unique prefix to identify a diagram test node.

id
Unique ID to identify a test node.

id_test
Unique test ID to identify a test node.

__init__(*prefix*, *config*, *object*)
Construct a test node (test) for any test objects (vms).

Parameters

- **name** (*str*) – name of the test node
- **config** (param.Reparsable) – variant configuration for the test node
- **object** (NetObject) – node-level object participating in the test node

__repr__()
Return repr(self).

static start_environment(*env_id*)
Start the environment for executing a test node.

Returns whether the environment is available after current or previous start

Return type bool

is_occupied()

is_scan_node()
Check if the test node is the root of all test nodes for all test objects.

is_terminal_node()
Check if the test node is the root of all test nodes for some test object.

is_shared_root()
Check if the test node is the root of all test nodes for all test objects.

is_object_root()
Check if the test node is the root of all test nodes for some test object.

is_objectless()
Check if the test node is not defined with any test object.

is_setup_ready(*worker*)
Check if all dependencies of the test were run or there were none.

Parameters **worker** (*str*) – relative setup readiness with respect to a worker ID

is_cleanup_ready(*worker*)
Check if all dependent tests were run or there were none.

Parameters **worker** (*str*) – relative setup readiness with respect to a worker ID

is_finished()
The test was run by at least one worker.

This choice of criterion makes sure that already available setup nodes are considered finished. If we instead wait for this setup to be cleaned up or synced, this would count most of the setup as finished in the end of the traversal while we would prefer to do so in an eager manner.

is_terminal_node_for()
Determine any object that this node is a root of.

Returns object that this node is a root of if any

Return type TestObject or None

produces_setup()
Check if the test node produces any reusable setup state.

Returns whether there are setup states to reuse from the test

Return type bool

has_dependency(*state*, *test_object*)
Check if the test node has a dependency parsed and available.

Parameters

- **state** (*str*) – name of the dependency (state or parent test set)
- **test_object** (TestObject) – object used for the dependency

Returns whether the dependency was already found among the setup nodes

Return type bool

classmethod prefix_priority(*prefix1*, *prefix2*)
Class method for secondary prioritization using test prefixes.

Parameters

- **prefix1** (*str*) – first prefix to use for the priority comparison
- **prefix2** (*str*) – second prefix to use for the priority comparison

This function also does recursive calls of sub-prefixes.

classmethod setup_priority(*node1*, *node2*)
Class method for setup traversal scheduling and prioritization.

Parameters

- **node1** (*TestNode*) – first node to use for the priority comparison
- **node2** (*TestNode*) – first node to use for the priority comparison

By default (if not externally set), it implements the divergent paths policy whereby workers will spread and explore the test space or equidistribute if confined within overlapping paths.

classmethod cleanup_priority(*node1*, *node2*)
Class method for cleanup traversal scheduling and prioritization.

Parameters

- **node1** (*TestNode*) – first node to use for the priority comparison
- **node2** (*TestNode*) – first node to use for the priority comparison

By default (if not externally set), it implements the divergent paths policy whereby workers will spread and explore the test space or equidistribute if confined within overlapping paths.

pick_parent(*slot*)
Pick the next available parent based on some priority.

Returns the next parent node

Return type *TestNode*

Raises RuntimeError

The current order will prioritize less traversed test paths.

pick_child(*slot*)
Pick the next available child based on some priority.

Returns the next child node

Return type *TestNode*

Raises RuntimeError

The current order will prioritize less traversed test paths.

visit_parent(*test_node*, *worker*)
Add a parent node to the set of visited nodes for this test.

Parameters

- **test_node** (*TestNode object*) – visited node
- **worker** (*str*) – slot ID of worker visiting the node

Raises ValueError if visited node is not directly dependent

visit_child(*test_node*, *worker*)
Add a child node to the set of visited nodes for this test.

Parameters

- **test_node** (*TestNode object*) – visited node
- **worker** (*str*) – slot ID of worker visiting the node

Raises ValueError if visited node is not directly dependent

add_location(*location*)
Add a setup reuse location information to the current node and its children.

Parameters **location** (*str*) – a special format string containing all information on the location where the format must be "gateway/host:path"

regenerate_params(*verbose=False*)
Regenerate all parameters from the current reparsable config.

Parameters **verbose** (*bool*) – whether to show generated parameter dictionaries

scan_states()
Scan for present object states to reuse the test from previous runs.

sync_states(*params*)
Sync or drop present object states to clean or later skip tests from previous runs.

validate()
Validate the test node for sane attribute-parameter correspondence.

avocado_i2n.cartgraph.object module

SUMMARY

Utility for the main test suite substructures like test objects.

Copyright: Intra2net AG

INTERFACE

class avocado_i2n.cartgraph.object.**TestObject**(*suffix*, *config*)
Bases: object

A wrapper for a test object used in one or more test nodes.

params
Parameters (cache) property.

final_restr
Final restriction to make the object parsing variant unique.

long_suffix
Sufficiently unique suffix to identify a variantless test object.

id
Unique ID to identify a test object.

__init__(*suffix*, *config*)
Construct a test object (vm) for any test nodes (tests).

Parameters

- **suffix** (*str*) – name of the test object
- **config** (param.Reparsable) – variant configuration for the test object

__repr__()
Return repr(self).

is_permanent()
If the test object is permanent, it can only be created manually (possibly through the use of manual setup steps).

On states on permanent test object are treated differently than on states on normal test object since they are preserved through test runs and even host shutdowns.

object_typed_params(*params*)
Return object and type filtered parameters using the current object type.

Parameters **params** (param_utils.Params) – whether to show generated parameter dictionaries

regenerate_params(*verbose=False*)
Regenerate all parameters from the current reparsable config.

Parameters **verbose** (*bool*) – whether to show generated parameter dictionaries

class avocado_i2n.cartgraph.object.**NetObject**(*name*, *config*)
Bases: *avocado_i2n.cartgraph.object.TestObject*

A Net wrapper for a test object used in one or more test nodes.

__init__(*name*, *config*)
Construct a test object (vm) for any test nodes (tests).

All arguments are inherited from the base class.

class avocado_i2n.cartgraph.object.**VMObject**(*name*, *config*)
Bases: *avocado_i2n.cartgraph.object.TestObject*

A VM wrapper for a test object used in one or more test nodes.

__init__(*name*, *config*)
Construct a test object (vm) for any test nodes (tests).

All arguments are inherited from the base class.

class avocado_i2n.cartgraph.object.**ImageObject**(*name*, *config*)
Bases: *avocado_i2n.cartgraph.object.TestObject*

An image wrapper for a test object used in one or more test nodes.

id
Sufficiently unique ID to identify a test object.

__init__(*name*, *config*)
Construct a test object (vm) for any test nodes (tests).

All arguments are inherited from the base class.

1.1.1.2 Module contents

1.1.2 avocado_i2n.plugins package

1.1.2.1 Submodules

avocado_i2n.plugins.auto module

class avocado_i2n.plugins.auto.**Auto**
Bases: avocado.core.plugin_interfaces.CLI

name = 'auto'

description = 'Autotesting using restriction-generated graph of setup state dependenci

configure(*parser*)
Add the subparser for the run action.

Parameters **parser** – Main test runner parser.

run(*config*)
Take care of command line overwriting, parameter preparation, setup and cleanup chains, and paths/utilities for all host controls.

__abstractmethods__ = frozenset()

avocado_i2n.plugins.manu module

class avocado_i2n.plugins.manu.**Manu**
Bases: avocado.core.plugin_interfaces.CLICmd

name = 'manu'

description = 'Tools using setup chains of manual steps with Cartesian graph manipulat

configure(*parser*)
Add the parser for the manual action.

Parameters **parser** – Main test runner parser.

run(*config*)
Take care of command line overwriting, parameter preparation, setup and cleanup chains, and paths/utilities for all host controls.

__abstractmethods__ = frozenset()

avocado_i2n.plugins.settings module

Avocado plugin that extends the settings path of our config paths.

class avocado_i2n.plugins.settings.**I2NSettings**
Bases: avocado.core.plugin_interfaces.Settings

adjust_settings_paths(*paths*)
Entry point where plugin can modify the list of configuration paths.

__abstractmethods__ = frozenset()

1.1.2.2 Module contents

1.1.3 avocado_i2n.states package

1.1.3.1 Submodules

avocado_i2n.states.btrfs module

SUMMARY

Module for the Btrfs state management backend.

Copyright: Intra2net AG

INTERFACE

class avocado_i2n.states.btrfs.**BtrfsBackend**
Bases: avocado_i2n.states.setup.StateBackend

Backend manipulating states as Btrfs volume snapshots.

classmethod show(*params*, *object=None*)
Return a list of available states of a specific type.

All arguments match the base class.

classmethod get(*params*, *object=None*)
Retrieve a state disregarding the current changes.

All arguments match the base class.

classmethod set(*params*, *object=None*)
Store a state saving the current changes.

All arguments match the base class.

classmethod unset(*params*, *object=None*)
Remove a state with previous changes.

All arguments match the base class.

classmethod check_root(*params*, *object=None*)
Check whether a root state or essentially the object exists.

All arguments match the base class.

classmethod unset_root(*params*, *object=None*)
Unset a root state to prevent object existence.

All arguments match the base class and in addition:

avocado_i2n.states.lvm module

SUMMARY

Module for the LVM state management backend.

Copyright: Intra2net AG

..warning:: **This state backend is mostly legacy and is not fully** maintained (only tested in isolation) but kept here for backwards portability as well as completeness in case there is interest to revive it by contributors that actually need it.

INTERFACE

class avocado_i2n.states.lvm.**LVMBackend**
Bases: avocado_i2n.states.setup.StateBackend

Backend manipulating states as logical volume snapshots.

classmethod show(*params*, *object=None*)
Return a list of available states of a specific type.

All arguments match the base class.

classmethod get(*params*, *object=None*)
Retrieve a state disregarding the current changes.

All arguments match the base class.

classmethod set(*params*, *object=None*)
Store a state saving the current changes.

All arguments match the base class.

classmethod unset(*params*, *object=None*)
Remove a state with previous changes.

All arguments match the base class and in addition:

Raises ValueError if LV pointer state was used

classmethod check_root(*params*, *object=None*)
Check whether a root state or essentially the object exists.

All arguments match the base class.

classmethod unset_root(*params*, *object=None*)
Unset a root state to prevent object existence.

All arguments match the base class and in addition:

Raises exceptions.TestWarn if permanent vm was detected

Remove the disk, virtual group, thin pool and logical volume of each object.

avocado_i2n.states.lvm.**vg_setup**(*vg_name*, *disk_vg_size*, *disk_basedir*, *disk_sparse_filename*, *use_tmpfs=True*)
Create volume group on top of ram memory to speed up LV performance.

When disk is specified the size of the physical volume is taken from existing disk space.

Parameters

- **vg_name** (*str*) – name of the volume group
- **disk_vg_size** (*str*) – size of the disk virtual group (MB)
- **disk_basedir** (*str*) – base directory for the disk sparse file
- **disk_sparse_filename** (*str*) – name of the disk sparse file
- **use_tmpfs** (*bool*) – whether to use RAM or slower storage

Returns disk_filename, vg_disk_dir, vg_name, loop_device

Return type (str, str, str, str)

Raises `lv_utils.LVException` on failure at any stage

Sample disk params: - disk_vg_size = "40000" - disk_basedir = "/tmp" - disk_sparse_filename = "virtual_hdd"

Sample general params: - vg_name='autotest_vg', - lv_name='autotest_lv', - lv_size='1G', - lv_snapshot_name='autotest_sn', - lv_snapshot_size='1G'

The disk volume group size is in MB.

`avocado_i2n.states.lvm.`**`vg_cleanup`**(*disk_filename=None*, *vg_disk_dir=None*, *vg_name=None*, *loop_device=None*, *use_tmpfs=True*)

Clean up any stage of the VG disk setup in case of test error.

This detects whether the components were initialized and if so tries to remove them. In case of failure it raises summary exception.

Parameters

- **`disk_filename`** (*str*) – name of the disk sparse file
- **`vg_disk_dir`** (*str*) – location of the disk file
- **`vg_name`** (*str*) – name of the volume group
- **`loop_device`** (*str*) – name of the disk or loop device
- **`use_tmpfs`** (*bool*) – whether to use RAM or slower storage

Returns disk_filename, vg_disk_dir, vg_name, loop_device

Return type (str, str, str, str)

Raises `lv_utils.LVException` on intolerable failure at any stage

avocado_i2n.states.lxc module

SUMMARY

Module for the LXC state management backend.

Copyright: Intra2net AG

INTERFACE

class `avocado_i2n.states.lxc.`**`LXCBackend`**

Bases: `avocado_i2n.states.setup.StateBackend`

Backend manipulating states as LXC container snapshots.

`classmethod show`(*params*, *object=None*)

Return a list of available states of a specific type.

All arguments match the base class.

`classmethod get`(*params*, *object=None*)

Retrieve a state disregarding the current changes.

All arguments match the base class.

classmethod set(*params*, *object=None*)
Store a state saving the current changes.

All arguments match the base class.

classmethod unset(*params*, *object=None*)
Remove a state with previous changes.

All arguments match the base class.

classmethod check_root(*params*, *object=None*)
Check whether a root state or essentially the object exists.

All arguments match the base class.

classmethod unset_root(*params*, *object=None*)
Unset a root state to prevent object existence.

All arguments match the base class and in addition:

avocado_i2n.states.pool module

SUMMARY

Module for the QCOW2 pool state management backend.

Copyright: Intra2net AG

INTERFACE

avocado_i2n.states.pool.**SKIP_LOCKS = False**
skip waiting on locks if we only read from the pool for all processes WARNING: use it only if you know what you are doing

class avocado_i2n.states.pool.**TransferOps**
Bases: object

A small namespace for pool transfer operations of multiple types.

classmethod list(*pool_path*, *params*)
List all states in a path from the pool.

Parameters

- **pool_path** (*str*) – pool path to list pool states from
- **params** (*{str, str}*) – configuration parameters

classmethod compare(*cache_path*, *pool_path*, *params*)
Compare cache and pool external state version.

Parameters

- **cache_path** (*str*) – cache path to compare with
- **pool_path** (*str*) – pool path to compare with
- **params** (*{str, str}*) – configuration parameters

classmethod download(*cache_path*, *pool_path*, *params*)
Download a path from the pool depending on the pool location.

Parameters

- **cache_path** (*str*) – cache path to download to
- **pool_path** (*str*) – pool path to download from
- **params** (*{str, str}*) – configuration parameters

classmethod upload(*cache_path*, *pool_path*, *params*)
Upload a path to the pool depending on the pool location.

Parameters

- **cache_path** (*str*) – cache path to upload from
- **pool_path** (*str*) – pool path to upload to
- **params** (*{str, str}*) – configuration parameters

classmethod delete(*pool_path*, *params*)
Delete a path in the pool depending on the pool location.

Parameters

- **pool_path** (*str*) – path in the pool to delete
- **params** (*{str, str}*) – configuration parameters

static list_local(*pool_path*, *params*)
List all states in a path from the pool.

All arguments are identical to the main entry method.

static compare_local(*cache_path*, *pool_path*, *params*)
Compare cache and pool external state version.

All arguments are identical to the main entry method.

static download_local(*cache_path*, *pool_path*, *params*)
Download a path from the pool depending on the pool location.

All arguments are identical to the main entry method.

static upload_local(*cache_path*, *pool_path*, *params*)
Upload a path to the pool depending on the pool location.

All arguments are identical to the main entry method.

static delete_local(*pool_path*, *params*)
Delete a path in the pool depending on the pool location.

All arguments are identical to the main entry method.

static list_remote(*pool_path*, *params*)
List all states in a path from the pool.

All arguments are identical to the main entry method.

static compare_remote(*cache_path*, *pool_path*, *params*)
Compare cache and pool external state version.

All arguments are identical to the main entry method.

static download_remote(*cache_path*, *pool_path*, *params*)
Download a path from the pool depending on the pool location.

All arguments are identical to the main entry method.

static upload_remote(*cache_path*, *pool_path*, *params*)
Upload a path to the pool depending on the pool location.

All arguments are identical to the main entry method.

static delete_remote(*pool_path*, *params*)
Delete a path in the pool depending on the pool location.

All arguments are identical to the main entry method.

static compare_link(*cache_path*, *pool_path*, *params*)
Compare cache and pool external state version.

All arguments are identical to the main entry method.

..todo:: True symlink support is available only for simple backing chains - we cannot have the same get_location for an entire chain here since some backing files are links and not the originals.

static list_link(*pool_path*, *params*)
List all states in a path from the pool.

All arguments are identical to the main entry method.

static download_link(*cache_path*, *pool_path*, *params*)
Download a path from the pool depending on the pool location.

All arguments are identical to the main entry method.

static upload_link(*cache_path*, *pool_path*, *params*)
Upload a path to the pool depending on the pool location.

All arguments are identical to the main entry method.

static delete_link(*pool_path*, *params*)
Delete a path in the pool depending on the pool location.

All arguments are identical to the main entry method.

class avocado_i2n.states.pool.**QCOW2ImageTransfer**
Bases: avocado_i2n.states.setup.StateBackend

Backend manipulating root or external states from a shared pool of QCOW2 images.

ops
alias of *TransferOps*

classmethod check_root(*params*, *object=None*)
Check whether a root state or essentially the object exists.

All arguments match the base class.

classmethod unset_root(*params*, *object=None*)
Unset a root state to prevent object existence.

All arguments match the base class and in addition:

classmethod compare_chain(*state*, *cache_dir*, *pool_dir*, *params*)
Compare checksums for all dependencies states backing a given state.

Parameters

- **state** (*str*) – state name
- **cache_dir** (*str*) – root cache directory to compare from/to
- **pool_dir** (*str*) – root pool directory to compare from/to

- **params** (*{str, str}*) – configuration parameters

classmethod transfer_chain(*state*, *cache_dir*, *pool_dir*, *params*, *down=True*)
Repeat pool operation an all dependencies states backing a given state.

Parameters

- **state** (*str*) – state name
- **cache_dir** (*str*) – root cache directory to transfer from/to
- **pool_dir** (*str*) – root pool directory to transfer from/to
- **params** (*{str, str}*) – configuration parameters
- **down** (*bool*) – whether the chain is downloaded or uploaded

classmethod show(*params*, *object=None*)
Return a list of available states of a specific type.

All arguments match the base class.

classmethod get(*params*, *object=None*)
Get a state transferring its entire chain of dependencies.

All arguments match the base class.

classmethod set(*params*, *object=None*)
Set a state transferring its entire chain of dependencies.

All arguments match the base class.

classmethod unset(*params*, *object=None*)
Unset a state preserving its entire chain of dependencies.

All arguments match the base class and in addition:

class avocado_i2n.states.pool.**RootSourcedStateBackend**
Bases: avocado_i2n.states.setup.StateBackend

Backend manipulating root states from a possibly shared source.

transport
alias of *QCOW2ImageTransfer*

classmethod check_root(*params*, *object=None*)
Check whether a root state or essentially the object exists.

All arguments match the base class.

classmethod unset_root(*params*, *object=None*)
Unset a root state to prevent object existence.

All arguments match the base class and in addition:

class avocado_i2n.states.pool.**SourcedStateBackend**
Bases: avocado_i2n.states.setup.StateBackend

Backend manipulating states from a possibly shared source.

transport
alias of *QCOW2ImageTransfer*

classmethod show(*params*, *object=None*)
Return a list of available states of a specific type.

All arguments match the base class.

classmethod get(*params*, *object=None*)
Get a state from the best possible mirror in a certain restricted scope.

All arguments match the base class.

classmethod set(*params*, *object=None*)
Set a state to all mirrors in a certain restricted scope.

All arguments match the base class.

classmethod unset(*params*, *object=None*)
Unset a state to all mirrors in a certain restricted scope.

All arguments match the base class and in addition:

avocado_i2n.states.pool.**image_lock**(*resource_path*, *timeout=300*)
Wait for a lock to free image for state pool operations.

Parameters

- **resource_path** (*str*) – path to the potentially locked resource
- **timeout** (*int*) – timeout to wait before erroring out (default 5 mins)

avocado_i2n.states.qcow2 module

SUMMARY

Module for the QCOW2 state management backends.

Copyright: Intra2net AG

INTERFACE

avocado_i2n.states.qcow2.**QEMU_OFF_STATES_REGEX = re.compile('^\\d+\\s+([\\w\\.-]+)\\s*(0 B)**
off qemu states regex (0 vm size)

avocado_i2n.states.qcow2.**QEMU_ON_STATES_REGEX = re.compile('^\\d+\\s+([\\w\\.-]+)\\s*(?!0 B**
on qemu states regex (>0 vm size)

class avocado_i2n.states.qcow2.**QCOW2Backend**
Bases: *avocado_i2n.states.pool.RootSourcedStateBackend*

Backend manipulating image states as internal QCOW2 snapshots.

classmethod state_type()
State type string representation depending used for logging.

classmethod show(*params*, *object=None*)
Return a list of available states of a specific type.

All arguments match the base class.

classmethod get(*params*, *object=None*)
Retrieve a state disregarding the current changes.

All arguments match the base class.

classmethod set(*params*, *object=None*)
Store a state saving the current changes.

All arguments match the base class.

classmethod unset(*params*, *object=None*)
Remove a state with previous changes.

All arguments match the base class.

class avocado_i2n.states.qcow2.**QCOW2ExtBackend**
Bases: *avocado_i2n.states.pool.SourcedStateBackend*, *avocado_i2n.states.qcow2.QCOW2Backend*

Backend manipulating image states as external QCOW2 snapshots.

classmethod check_root(*params*, *object=None*)
Check whether a root state or essentially the object exists locally.

All arguments match the base class.

classmethod unset_root(*params*, *object=None*)
Unset a root state to prevent object existence.

All arguments match the base class.

class avocado_i2n.states.qcow2.**QCOW2VTBackend**
Bases: *avocado_i2n.states.qcow2.QCOW2Backend*

Backend manipulating vm states as QCOW2 snapshots using VT's VM bindings.

classmethod show(*params*, *object=None*)
Return a list of available states of a specific type.

All arguments match the base class.

classmethod get(*params*, *object=None*)
Retrieve a state disregarding the current changes.

All arguments match the base class.

classmethod set(*params*, *object=None*)
Store a state saving the current changes.

All arguments match the base class.

classmethod unset(*params*, *object=None*)
Remove a state with previous changes.

All arguments match the base class.

avocado_i2n.states.qcow2.**get_image_path**(*params*)
Get the absolute path to a QCOW2 image.

Parameters **params** (*{str, str}*) – configuration parameters

Returns absolute path to the QCOW2 image

Return type str

avocado_i2n.states.qcow2.**convert_image**(*params*)
Convert a raw img to a QCOW2 or other image usable for virtual machines.

Parameters **params** (*{str, str}*) – configuration parameters

Raises py:class:*FileNotFoundError* if the source image doesn't exist

Raises py:class:*AssertionError* when the source image is in use

Note: This function could be used with qemu-img for more general images and not just the QCOW2 format.

avocado_i2n.states.ramfile module

SUMMARY

Module for the ramfile state management backend.

Copyright: Intra2net AG

INTERFACE

class avocado_i2n.states.ramfile.**RamfileBackend**
Bases: *avocado_i2n.states.pool.SourcedStateBackend*

Backend manipulating vm states as ram dump files.

image_state_backend = None

classmethod check_root(*params*, *object=None*)
Check whether a root state or essentially the object is running.

All arguments match the base class.

classmethod unset_root(*params*, *object=None*)
Unset a root state to prevent object from running.

All arguments match the base class.

avocado_i2n.states.setup module

SUMMARY

Utility to manage off and on test object states.

Copyright: Intra2net AG

INTERFACE

avocado_i2n.states.setup.**BACKENDS = {}**
available state backend implementations

avocado_i2n.states.setup.**ROOTS = ['root', '0root', 'boot', '0boot']**
keywords reserved for root states

avocado_i2n.states.setup.**show_states**(*run_params*, *env=None*)
Return a list of available states of a specific type.

Parameters

- **run_params** (*{str, str}*) – configuration parameters
- **env** (*Env object or None*) – test environment or nothing if not needed

Returns list of detected states

Return type [str]

avocado_i2n.states.setup.**check_states**(*run_params*, *env=None*)
Check whether a given state exits.

Parameters **run_params** (*{str, str}*) – configuration parameters

Returns whether the given state exists

Return type bool

Note: We can check for multiple states of multiple objects at the same time through our choice of configuration.

avocado_i2n.states.setup.**get_states**(*run_params*, *env=None*)
Retrieve a state disregarding the current changes.

Parameters **run_params** (*{str, str}*) – configuration parameters

Returns list of detected states

Raises exceptions.TestAbortError if the retrieved state doesn't exist, the vm is unavailable from the env, or snapshot exists in passive mode (abort)

Raises exceptions.TestError if invalid policy was used

avocado_i2n.states.setup.**set_states**(*run_params*, *env=None*)
Store a state saving the current changes.

Parameters **run_params** (*{str, str}*) – configuration parameters

Returns list of detected states

Raises exceptions.TestAbortError if unexpected/missing snapshot in passive mode (abort)

Raises exceptions.TestError if invalid policy was used

avocado_i2n.states.setup.**unset_states**(*run_params*, *env=None*)
Remove a state with previous changes.

Parameters **run_params** (*{str, str}*) – configuration parameters

Returns list of detected states

Raises exceptions.TestAbortError if missing snapshot in passive mode (abort)

Raises exceptions.TestError if invalid policy was used

avocado_i2n.states.setup.**push_states**(*run_params*, *env=None*)
Identical to the set operation but used within the push/pop pair.

Parameters **run_params** (*{str, str}*) – configuration parameters

Returns list of detected states

avocado_i2n.states.setup.**pop_states**(*run_params*, *env=None*)
Retrieve and remove a state/snapshot.

Parameters **run_params** (*{str, str}*) – configuration parameters

Returns list of detected states

avocado_i2n.states.vmnet module

SUMMARY

Module for the VMNet state management backend.

Copyright: Intra2net AG

INTERFACE

class avocado_i2n.states.vmnet.**VMNetBackend**
Bases: avocado_i2n.states.setup.StateBackend

Backend manipulating network states as VMNet operations.

network_class
alias of *avocado_i2n.vmnet.network.VMNetwork*

classmethod show(*params*, *object=None*)
Return a list of available states of a specific type.

All arguments match the base class.

classmethod get(*params*, *object=None*)
Retrieve a state disregarding the current changes.

All arguments match the base class.

classmethod set(*params*, *object=None*)
Store a state saving the current changes.

All arguments match the base class.

classmethod unset(*params*, *object=None*)
Remove a state with previous changes.

All arguments match the base class.

classmethod check_root(*params*, *object=None*)
Check whether a root state or essentially the object exists.

All arguments match the base class.

classmethod unset_root(*params*, *object=None*)
Unset a root state to prevent object existence.

All arguments match the base class and in addition:

1.1.3.2 Module contents

1.1.4 avocado_i2n.vmnet package

1.1.4.1 Submodules

avocado_i2n.vmnet.interface module

SUMMARY

Interface object for the vmnet utility.

Copyright: Intra2net AG

CONTENTS

This is the basic building block of the vm network. Interfaces are grouped in nodes (the virtual machines they belong to) and in netconfigs (the local networks they define together).

INTERFACE

class avocado_i2n.vmnet.interface.**VMInterface**(*name*, *params*)
Bases: object

The interface class.

node
A reference to the node the interface belongs to.

netconfig
A reference to the netconfig the interface belongs to.

params
Configuration properties

mac
MAC address used by the network interface.

ip
Interface properties

name
Name for the interface.

__init__(*name*, *params*)
Construct an interface with configuration from the parameters.

Parameters **params** (*{str, str}*) – configuration parameters

__repr__()
Return repr(self).

avocado_i2n.vmnet.netconfig module

SUMMARY

Network configuration object for the VM network.

Copyright: Intra2net AG

CONTENTS

It contains the network configuration, offers network services like IP address allocation, translation, and validation, and consists of Interface objects that share this network configuration.

INTERFACE

class avocado_i2n.vmnet.netconfig.**VMNetconfig**
Bases: object

The netconfig class - a collection of interfaces sharing the same network configuration.

interfaces
Configuration properties

netdst
The bridge where Qemu will redirect the packets.

Plays the role of the network connectivity skeleton.

netmask
The netmask used by the participating network interfaces.

mask_bit
The netmask bit used by the participating network interfaces.

gateway
The gateway ip used by the participating network interfaces.

net_ip
The network ip used by the participating network interfaces.

host_ip
IP of the host for the virtual machine if it participates in the local network (and therefore in the netcofig).

range
IP range of addresses that can be allocated to joining vms (new interfaces that join the netconfig).

To set a different ip_start and ip_end, i.e. different boundaries, use the setter of this property.

Note: Used for any DHCP configuration.

---

ip_start
Beginning of the IP range.

ip_end
End of the IP range.

domain
DNS domain name for the local network.

Note: Used for host-based DNS configuration.

---

forwarder
DNS forwarder address for the local network.

Note: Used for host-based DNS configuration.

---

rev
DNS reverse lookup table name for the local network.

Note: Used for host-based DNS configuration.

---

view
DNS view name for the local network.

Note: Used for host-based DNS configuration.

ext_netdst
External network destination to which we route after network translation.

Note: Used for host-based NAT configuration.

__init__()
Construct a nonconfigured netconfig.

__repr__()
Return repr(self).

from_interface(*interface*)
Construct all netconfig parameters from the provided interface or reset them with respect to that interface if they were already set.

Parameters **interface** (*Interface object*) – reference interface for the configuration

add_interface(*interface*)
Add an interface to the netconfig, performing the necessary registrations and finishing with validation of the interface configuration.

Parameters **interface** (*Interface object*) – interface to add to the netconfig

has_interface(*interface*)
Check whether an interface already belongs to the netconfig through both IP and actual attachment (to counter same IP range netconfigs).

Parameters **interface** (*Interface object*) – interface to check in the netconfig

Returns whether the interface is already present in the netconfig

Return type bool

can_add_interface(*interface*)
Check if an interface can be added to the netconfig based on its desired IP address and throw Exceptions if it is already present or the netmask does not coincide (misconfiguration errors).

Parameters **interface** (*Interface object*) – interface to add to the netconfig

Returns whether the interface can be added

Return type bool

Raises exceptions.IndexError if interface is already present or incompatible

validate()
Check for sanity of the netconfigs parameters.

Raises exceptions.TestError if the validation fails

translate_address(*ip*, *nat_ip*)
Return the NAT translated IP of an interface or alternatively the IP of an interface masked by a desired network address.

Parameters

- **interface** (*Interface object*) – interface to translate
- **nat_ip** (*str*) – NATed IP to use for reference

Returns the translated IP of the interface

Return type str

avocado_i2n.vmnet.network module

SUMMARY

Utility to manage local networks of vms and various topologies.

Copyright: Intra2net AG

CONTENTS

The main class is the VMNetwork class and is used to perform all network related configuration for each virt test (store all its network information, offer all the network related services it needs, etc.). It can be used to test Proxy, Port forwarding, VPN, NAT, etc.

Each vm is a network node and can have one of few currently supported operating systems. For ease of defaults use it is recommended to have at least three nics, respectively with the role of host nic for local isolated connection to the host, the role of internet nic for (internet) connection to the other nodes, and the role of LAN nic for other any other connections to vm's own LANs.

Ephemeral clients are based on RIP Linux and are temporary clients created just for the duration of a test. An arbitrary number of those can be spawned depending on test requirements and available resources.

INTERFACE

avocado_i2n.vmnet.network.**BIND_DHCP_CONFIG = '/etc/dhcp/dhcpd.conf'**
networking service backend paths

class avocado_i2n.vmnet.network.**VMNetwork**(*params*, *env*)
Bases: object

Any VMNetwork instance can be used to connect vms in various network topologies and to reconfigure, ping, retrieve the session of, as well as spawn clients for each of them.

__init__(*params*, *env*)
Construct a network data structure given the test parameters, the *env* and the *test* instance.

Note: The params attribute is just a shallow copy to preserve the hierarchy: network level params = test level params -> node level params = test object params -> interface level params = rarely used outside of the vm network

---

__repr__()
Return repr(self).

start_all_sessions()
Start a session to each of the vm nodes.

integrate_node(*node*)
Add all interfaces and netconfigs resulting from a new vm node into the vm network, thus integrating the configuration into the available one.

Parameters **node** (VMNode) – vm node to be integrated into the network

reattach_interface(*client*, *server*, *client_nic='internet_nic'*, *server_nic='lan_nic'*, *proxy_nic=''*)

Reconfigure a network interface of a vm reattaching it to a different interface's network config.

Parameters

- **client** (virttest.qemu_vm.VM) – vm whose interace will be rattached
- **server** (virttest.qemu_vm.VM) – vm whose network will the interface be attached to
- **client_nic** (*str*) – role of the nic of the client
- **server_nic** (*str*) – role of the nic of the server
- **proxy_nic** (*str*) – name of a proxyARP nic of the server

If the *proxy_nic* is defined and the second interface (*server_nic*) is different than the *proxy_nic* value, it is assumed to be and turned into a proxyarp interface whose responses are provided by the actual interface defined in the *proxy_nic* parameter.

Any processing related to the vm or the netconfig's servers must be performed separately.

A typical processing for the clients is to insert a DHCP/DNS host, while a typical processing for the vm is to recreate it on top of the moved bridges with or without session.

A typical processing for the vm is to reconfigure the nic type of *server_nic* to PROXYARP and its IP to the IP of *proxy_nic*.

setup_host_services()

Provide all necessary services like DHCP, DNS and NAT to restrict all tests locally.

setup_host_bridges()

Setup bridges and interfaces needed to create and isolate the network.

The final network topology is derived entirely from the test parameters.

spawn_clients(*server_name*, *clients_num*, *nic='lan_nic'*)

Create and boot ephemeral clients for a given server.

Parameters

- **server_name** (*str*) – name of the vm that plays the role of a server
- **clients_num** (*int*) – number of ephemeral clients to spawn
- **nic** (*str*) – name of the nic of the server

Returns generated ephemeral clients

Return type (virttest.qemu_vm.VM)

change_network_address(*netconfig*, *new_ip*, *new_mask=None*)

Change the ip of a netconfig and more specifically of the network interface of any vm participating in it.

Parameters

- **netconfig** (VMNetconfig) – netconfig to change the IP of
- **new_ip** (*str*) – new IP address for the netconfig
- **new_mask** (*str or None*) – new network mask for the netconfig

Note: The network must have at least one interface in order to change its address.

configure_tunnel_between_vms(*name*, *vm1*, *vm2*, *local1=None*, *remote1=None*, *peer1=None*, *auth=None*, *apply_extra_options=None*)

Configure a tunnel between two vms.

Parameters

- **name** (*str*) – name of the tunnel
- **vm1** (virttest.qemu_vm.VM) – left side vm of the tunnel
- **vm2** (virttest.qemu_vm.VM) – right side vm of the tunnel
- **local1** (*{str, str}*) – left local type as in tunnel constructor
- **remote1** (*{str, str}*) – left remote type as in tunnel constructor
- **peer1** (*{str, str}*) – left peer type as in tunnel constructor
- **auth** (*{str, str}*) – authentication configuration as described in the tunnel constructor
- **apply_extra_options** (*{str, any}*) – extra switches to apply as key exchange, firewall ruleset, etc.

configure_tunnel_on_vm(*name*, *vm*, *apply_extra_options=None*)

Configure a tunnel on a vm, assuming it is manually or independently configured on the other end.

Parameters

- **name** (*str*) – name of the tunnel
- **vm** (virttest.qemu_vm.VM) – vm where the tunnel will be configured
- **apply_extra_options** (*{str, any}*) – extra switches to apply as key exchange, firewall ruleset, etc.

Raises exceptions.KeyError if not all tunnel parameters are present

Currently the method uses only existing tunnels.

configure_roadwarrior_vpn_on_server(*name*, *server*, *client*, *local1=None*, *remote1=None*, *peer1=None*, *auth=None*, *apply_extra_options=None*)

Configure a VPN connection (tunnel) on a vm to play the role of a VPN server for any individual clients to access it from the internet.

Arguments are similar to the ones from *configure_tunnel_between_vms()* with the exception of:

Parameters

- **server** (virttest.qemu_vm.VM) – vm which will be the VPN server for roadwarrior connections
- **client** (virttest.qemu_vm.VM) – vm which will be connecting individual device

Regarding the client, only its parameters will be updated by this method.

configure_vpn_route(*vms*, *vpns*, *remote1=None*, *peer1=None*, *auth=None*, *extra_apply_options=None*)

Build a set of VPN connections using VPN forwarding to gain access from one vm to another.

Arguments are similar to the ones from *configure_tunnel_between_vms()* with the exception of:

Parameters

- **vms** ([virttest.qemu_vm.VM]) – vms to participate in the VPN route
- **vpns** (*[str]*) – VPNs over which the route will be constructed

Raises exceptions.TestError if #vpns < #vms - 1 or #vpns < 2 or #vms < 2

Infrastructure of point to point VPN connections must already exist.

verify_vpn_in_log(*src_vm*, *dst_vm*, *log_vm=None*, *require_blocked=False*)
Search for the appropriate message in the vpn log file.

Parameters

- **src_vm** (virttest.qemu_vm.VM) – source vm whose packets will be logged
- **dst_vm** (virttest.qemu_vm.VM) – destination vm whose packets will be logged
- **log_vm** (virttest.qemu_vm.VM) – vm where all packets are logged
- **require_blocked** (*bool*) – whether to expect access message or deny message

Raises exceptions.TestError if the source or destination vms are not on the network

Raises exceptions.TestFail if the VPN packets were not logged properly

This function requires modified firewall ruleset for the vpn connection.

ping(*src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*)
Pings a vm from another vm to test basic ICMP connectivity.

Parameters

- **src_vm** (virttest.qemu_vm.VM) – source vm which will ping
- **dst_vm** (virttest.qemu_vm.VM) – destination vm which will be pinged
- **dst_nic** (*str*) – nic of the destination vm used if necessary to obtain accessible IP
- **address** (*str*) – explicit IP or domain to use for pinging

Returns the status and output of the performed ping

Return type (int, str)

If no *address* is provided, the IP is obtained by analyzing the network topology from *src_vm* to *dst_vm*.

If no *dst_vm* is provided, the ping happens directly to *address*.

ping_validate(*src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *timeout=30*)
Pings a vm from another vm to test basic ICMP connectivity and bails on nonzero status.

Arguments are similar to the ones from *ping()* with the exception of:

Parameters **timeout** (*int*) – number of seconds to retry the ping for as networking might not be immediately available

Raises exceptions.TestError if the performed ping failed

This method does not perform a refined exit status check, you can use the non-validated version and perform your own customization if you wish.

ping_all(*timeout=30*)
Pings all nodes from each other in order to test complete basic ICMP connectivity.

Parameters **timeout** (*int*) – number of seconds to retry the ping for as networking might not be immediately available

Raises exceptions.TestError if a network mutual ping failed

The ping happens among all LAN members, throwing an exception if one of the pings fails.

port_connectivity(*msg*, *src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=80*, *protocol='TCP'*)

Test connectivity using a predefined port (usually in addition to pinging).

Arguments are similar to the *ping()* method with the exception of:

Parameters

- **msg** (*str*) – probing data to be sent to the port
- **port** (*int*) – forwarding port to send the message to
- **protocol** (*str*) – protocol type (TCP, UDP or something over them)

Returns the result of the performed port connection attempt

Return type (int, str)

port_connectivity_validate(*msg*, *src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=80*, *protocol='TCP'*, *validate_output=''*, *require_blocked=False*)

Test connectivity using a predefined port (usually in addition to pinging).

Arguments are similar to the *port_connectivity()* method with the exception of:

Parameters

- **validate_output** (*str*) – string to find in the command output and validate against
- **require_blocked** (*bool*) – whether to expect nonzero status from the connection attempt

Raises exceptions.TestError if the performed port connection attempt failed

This method does not perform a refined exit status check, you can use the non-validated version and perform your own customization if you wish.

http_connectivity(*src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=80*, *protocol='HTTP'*)

Test connectivity using an HTTP port and protocol.

Arguments are similar to the *port_connectivity()* method.

Raises exceptions.TestError if inappropriate protocol was given

http_connectivity_validate(*src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=80*, *protocol='HTTP'*, *require_blocked=False*)

Test connectivity using an HTTP port and protocol.

Arguments are similar to the *port_connectivity()* method.

Raises exceptions.TestError if inappropriate protocol was given

https_connectivity(*src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=443*, *protocol='HTTPS'*)

Test connectivity using an HTTPS port and protocol.

Arguments are similar to the *port_connectivity()* method.

Raises exceptions.TestError if inappropriate protocol was given

https_connectivity_validate(*src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=443*, *protocol='HTTPS'*, *require_blocked=False*)

Test connectivity using an HTTPS port and protocol.

Arguments are similar to the *port_connectivity()* method.

Raises `exceptions.TestError` if inappropriate protocol was given

ssh_connectivity(*src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=22*, *protocol='SSH'*)
Test connectivity using an SSH port and protocol.

Arguments are similar to the *port_connectivity()* method.

Raises `exceptions.TestError` if inappropriate protocol was given

ssh_connectivity_validate(*src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=22*, *protocol='SSH'*, *require_blocked=False*)
Test connectivity using an SSH port and protocol.

Arguments are similar to the *port_connectivity()* method.

Raises `exceptions.TestError` if inappropriate protocol was given

ssh_hostname(*src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *timeout=10*)
Get the host name of a vm from any other vm in the vm net using the SSH protocol.

Parameters

- **src_vm** (`virttest.qemu_vm.VM`) – source vm with the SSH client
- **dst_vm** (`virttest.qemu_vm.VM`) – destination vm with the SSH server
- **dst_nic** (*str*) – nic of the destination vm used if necessary to obtain accessible IP
- **timeout** (*int*) – timeout for the SSH connection

Returns the hostname of the SSH server

Return type str+

This tests the TCP connectivity and verifies it leads to the correct machine.

scp_files(*src_path*, *dst_path*, *src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *timeout=10*)
Copy files securely where built-in methods like `vm.copy_files_to()` fail.

Parameters

- **src_path** (*str*) – source path for the securely copied files
- **dst_path** (*str*) – destination path for the securely copied files
- **src_vm** (`virttest.qemu_vm.VM`) – source vm with the ssh client
- **dst_vm** (`virttest.qemu_vm.VM`) – destination vm with the ssh server
- **dst_nic** (*str*) – nic of the destination vm used if necessary to obtain accessible IP
- **timeout** (*int*) – timeout for the SSH connection

Raises `exceptions.TestFail` if the files couldn't be copied

The paths *src_path* and *dst_path* must be strings, possibly with a wildcard.

ftp_connectivity(*msg*, *file*, *src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=21*)
Send file request to an FTP destination port and address and verify it was received.

Arguments are similar to the *port_connectivity()* method with the exception of:

Parameters **file** (*str or None*) – file to retrieve containing the test data or none if sent directly

Raises `exceptions.TestError` if inappropriate protocol was given

ftp_connectivity_validate(*msg*, *file*, *src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=21*, *require_blocked=False*)
Send file request to an FTP destination port and address and verify it was received.

Arguments are similar to the *port_connectivity()* method with the exception of:

Parameters **file** (*str or None*) – file to retrieve containing the test data or none if sent directly

Raises exceptions.TestError if inappropriate protocol was given

tftp_connectivity(*msg*, *file*, *src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=69*)
Send file request to an TFTP destination port and address and verify it was received. Arguments are similar to the *port_connectivity()* method with the exception of:

Parameters **file** (*str or None*) – file to retrieve containing the test data or none if sent directly

Raises exceptions.TestError if inappropriate protocol was given

tftp_connectivity_validate(*msg*, *file*, *src_vm*, *dst_vm*, *dst_nic='lan_nic'*, *address=None*, *port=69*, *require_blocked=False*)
Send file request to an TFTP destination port and address and verify it was received. Arguments are similar to the *port_connectivity()* method with the exception of:

Parameters **file** (*str or None*) – file to retrieve containing the test data or none if sent directly

Raises exceptions.TestError if inappropriate protocol was given

avocado_i2n.vmnet.node module

SUMMARY

VMNode object for the vmnet utility.

Copyright: Intra2net AG

CONTENTS

This class wraps up the functionality shared among the interfaces of the same platform like session management, etc.

INTERFACE

class avocado_i2n.vmnet.node.**VMNode**(*platform*, *ephemeral=False*)
Bases: object

The vmnode class - a collection of interfaces sharing the same platform.

interfaces
A collection of interfaces the vm node represents.

ephemeral
Platform properties

platform
A reference to the virtual machine object whose network configuration is represented by the vm node.

name
A proxy reference to the vm name.

params
A proxy reference to the vm params.

Note: this is just a shallow copy to preserve the hierarchy: network level params = test level params -> vmnode level params = test object params -> interface level params = rarely used outside of the vm network

---

remote_sessions
A proxy reference to the vm sessions.

last_session
A pointer to the last created vm session.

Used to facilitate the frequent access to a single session.

__init__ (*platform*, *ephemeral=False*)
Construct a vm node from a vm platform.

Parameters

- **platform** (`virttest.qemu_vm.VM`) – the vm platform that communicates in the vm network
- **ephemeral** (*bool*) – whether the node is ephemeral (spawned in a network)

__repr__ ()
Return repr(self).

check_interface (*condition*)
Check whether one of node's interfaces satisfies a boolean condition.

Parameters **condition** (*function*) – condition to try each interface on

Returns the first interface satisfying the provided criteria or None

Return type Interface object or None

avocado_i2n.vmnet.tunnel module

SUMMARY

Tunnel object for the vmnet utility.

Copyright: Intra2net AG

CONTENTS

This class wraps up the utilities for managing tunnels.

The parameters parsed at each vm are used as overwrite dictionary and and missing ones are generated for the full configuration of the tunnel.

INTERFACE

class avocado_i2n.vmnet.tunnel.**VMTunnel**(*name*, *node1*, *node2*, *local1=None*, *remote1=None*, *peer1=None*, *auth=None*)

Bases: object

The tunnel class.

left
A reference to the left node of the tunnel.

right
A reference to the right node of the tunnel.

left_iface
A reference to the left interface of the tunnel.

right_iface
A reference to the right interface of the tunnel.

left_net
A reference to the left netconfig of the tunnel.

right_net
A reference to the right netconfig of the tunnel.

left_params
The tunnel generated left side parameters.

right_params
The tunnel generated right side parameters.

params
Connection properties

name
Name for the connection.

__init__(*name*, *node1*, *node2*, *local1=None*, *remote1=None*, *peer1=None*, *auth=None*)
Construct the full set of required tunnel parameters for a given tunnel left configuration that are not already defined in the parameters of the two vms (left *node1* with right *node2*).

Parameters

- **name** (*str*) – name of the tunnel
- **node1** (*VMNode object*) – left side node of the tunnel
- **node2** (*VMNode object*) – right side node of the tunnel
- **local1** (*{str, str}*) – left local configuration with at least one key 'type' with value 'nic' for left-site (could be used for site-to-site or site-to-point tunnels) or 'internetip' for left-point (for point-to-site or point-to-point tunnels) or 'custom' for left-site or left-point that is not a LAN (e.g. for tunnel forwarding of another tunneled remote net)
- **remote1** (*{str, str}*) – left remote configuration with at least one key 'type' with value 'custom' for right-site (could be used for site-to-site or point-to-site tunnels) or 'externalip' for right-point (for site-to-point or point-to-point tunnels) or 'modeconfig' for special right-point (using a ModeConfig connection for a right road warrior)
- **peer1** (*{str, str}*) – left peer configuration with at least one key 'type' with value 'ip' for no NAT along the tunnel (the peer having a public IP) or 'dynip' for a road warrior right end point (the peer is behind NAT and its IP is changing)

- **auth** (*{str, str}*) – authentication configuration with at least one key 'type' with value in "pubkey", "psk", "none" and the rest of the keys providing type details

Raises ValueError if some of the supplied configuration is not valid

The right side *local2*, *remote2*, *peer2* configuration is determined from the left side.

If a PSK (pre-shared secret) authentication type is specified, the relevant additional options are *psk* for the secret word, *left_id* and *right_id* for the identification type to be used on each side (either IP for empty id or any user-defined id).

__repr__()
Return repr(self).

connects_nodes(*node1*, *node2*)
Check whether a tunnel connects two vm nodes, i.e. they are in directly connected as tunnel peers or indirectly in tunnel connected LANs (netconfigs).

Parameters

- **node1** (*VM node*) – one side vm of the tunnel
- **node2** (*VM node*) – another side vm of the tunnel

Returns whether the tunnel connects the two nodes

Return type bool

configure_between_endpoints(*apply_extra_options=None*)
Build a tunnel between two endpoint vms.

Parameters **apply_extra_options** (*{str, any}*) – extra switches to apply as key exchange, firewall ruleset, etc.

configure_on_endpoint(*node*, *apply_extra_options=None*)
Configure a tunnel on an end point virtual machine.

Parameters

- **node** (*VMNode object*) – node end point where the tunnel will be configured
- **apply_extra_options** (*{str, any}*) – extra switches to apply as key exchange, firewall ruleset, etc.

Raises ValueError if some of the supplied configuration is not valid

The provided virtual machine parameters will be used for configuration of the tunnel.

The tunnel name can be used to also reconfigure an existing tunnel.

import_key_params(*from_node*, *to_node*)
This will generate own key configuration at the source vm and foreign key configuration at the destination vm.

Parameters

- **from_node** (*VMNode object*) – source node to get the key from (and generate own key configuration on it containing all relevant key information)
- **to_node** (*VMNode object*) – destination node to import the key to (and generate foreign key configuration on it containing all relevant key information)

1.1.4.2 Module contents

1.2 Submodules

1.2.1 avocado_i2n.cmd_parser module

avocado_i2n.cmd_parser.**params_from_cmd**(*config*)

Take care of command line overwriting, parameter preparation, setup and cleanup chains, and paths/utilities for all host controls.

Parameters **config** (*{str, str}*) – command line arguments

Raises ValueError if a command line selected vm is not available from the configuration and thus supported or internal tests are restricted from the command line

---

Todo: Any dynamically created config keys here are usually entire data structures like dictionaries and lists and only used internally during the run which makes them unfit for displaying to the user and putting in a namespace scope like the officially registered plugin settings. Let's wait to see if the multi-suite support in avocado would establish some standards for doing this first. Until then, the user won't directly interact with these keys anyway.

---

avocado_i2n.cmd_parser.**full_vm_params_and_strs**(*param_dict*, *vm_strs*, *use_vms_default*)

Add default vm parameters and strings for missing command line such.

Parameters

- **param_dict** (*{str, str} or None*) – runtime parameters used for extra customization
- **vm_strs** (*{str, str}*) – command line vm-specific names and variant restrictions
- **use_vms_default** (*{str, bool}*) – whether to use default variant restriction for a particular vm

Returns complete vm parameters and strings

Return type (Params, {str, str})

Raises ValueError if no command line or default variant restriction could be found for some vm

avocado_i2n.cmd_parser.**full_tests_params_and_str**(*param_dict*, *tests_str*, *use_tests_default*)

Add default tests parameters and string for missing command line such.

Parameters

- **param_dict** (*{str, str} or None*) – runtime parameters used for extra customization
- **tests_str** (*str*) – command line variant restrictions
- **use_tests_default** (*bool*) – whether to use default primary restriction

Returns complete tests parameters and string

Return type (Params, str)

Raises ValueError if the default primary restriction could is not valid (among the available ones)

avocado_i2n.cmd_parser.**env_process_hooks**()

Add env processing hooks to handle on/off state get/set operations and vmnet networking setup and instance attachment to environment.

1.2.2 avocado_i2n.intertest_setup module

1.2.2.1 SUMMARY

Utility to manage all needed virtual machines.

1.2.2.2 CONTENTS

This utility can be used by any host control to manage one or more virtual machines. It in turn uses some other host utilities.

Use the tag argument to add more details to generated test variant name in case you are running any of the manual step functions here more than once.

IMPORTANT: If you don't want to perform the given setup with all virtual machines, defined by your parameters then just overwrite the parameter *vms* as a space separated list of the selected virtual machine names. The setup then is going to be performed only on those machines and not on all. Example is 'vms = vm1 vm2 vm3 ' to create only vm1 and vm3 add to the overwrite string 'vms = vm1 vm3 ' in order to overwrite the vms parameter. Of course you can do this with any parameter to manage other aspects of the virtual environment setup process.

1.2.2.3 INTERFACE

avocado_i2n.intertest_setup.**noop**(*config*, *tag=''*)
Empty setup step to invoke plugin without performing anything.

Parameters

- **config** (*{str, str}*) – command line arguments and run configuration
- **tag** (*str*) – extra name identifier for the test to be run

avocado_i2n.intertest_setup.**unittest**(*config*, *tag=''*)
Perform self testing for sanity and test result validation.

Parameters

- **config** (*{str, str}*) – command line arguments and run configuration
- **tag** (*str*) – extra name identifier for the test to be run

avocado_i2n.intertest_setup.**full**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**update**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**run**(*config*, *tag=''*)
Run a set of tests without any automated setup.

Parameters

- **config** (*{str, str}*) – command line arguments and run configuration
- **tag** (*str*) – extra name identifier for the test to be run

This is equivalent to but more powerful than the runner plugin.

avocado_i2n.intertest_setup.**list**(*config*, *tag=''*)
List a set of tests from the command line.

Parameters

- **config** (*{str, str}*) – command line arguments and run configuration
- **tag** (*str*) – extra name identifier for the test to be run

This is equivalent to but more powerful than the loader plugin.

avocado_i2n.intertest_setup.**install**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**deploy**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**internal**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**boot**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**download**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**upload**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**shutdown**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**check**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**pop**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**push**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**get**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**set**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**unset**(*config*, *tag=''*)

avocado_i2n.intertest_setup.**collect**(*config*, *tag=''*)

Get a new test object (vm, root state) from a pool.

Parameters

- **config** (*{str, str}*) – command line arguments and run configuration
- **tag** (*str*) – extra name identifier for the test to be run

..todo:: With later refactoring of the root check implicitly getting a pool rool state, we can refine the parameters here.

avocado_i2n.intertest_setup.**create**(*config*, *tag=''*)

Create a new test object (vm, root state).

Parameters

- **config** (*{str, str}*) – command line arguments and run configuration
- **tag** (*str*) – extra name identifier for the test to be run

avocado_i2n.intertest_setup.**clean**(*config*, *tag=''*)

Remove a test object (vm, root state).

Parameters

- **config** (*{str, str}*) – command line arguments and run configuration
- **tag** (*str*) – extra name identifier for the test to be run

1.2.3 avocado_i2n.loader module

1.2.3.1 SUMMARY

Specialized test loader for the plugin.

Copyright: Intra2net AG

1.2.3.2 INTERFACE

class avocado_i2n.loader.**CartesianLoader**(*config=None*, *extra_params=None*)
Bases: avocado.core.plugin_interfaces.Resolver

Test loader for Cartesian graph parsing.

name = 'cartesian_graph'

description = 'Loads tests by Cartesian graph parsing'

__init__(*config=None*, *extra_params=None*)
Construct the Cartesian loader.

Parameters

- **config** (*{str, str}*) – command line arguments
- **extra_params** (*{str, str}*) – extra configuration parameters

parse_object_variants(*param_dict=None*, *object_strs=None*, *verbose=False*)
Parse composite test object variants from joined component variants.

Parameters

- **param_dict** (*{str, str} or None*) – runtime parameters used for extra customization
- **object_strs** (*{str, str}*) – object-specific names and variant restrictions
- **verbose** (*bool*) – whether to print extra messages or not

Returns parsed test objects

Return type [TestObject]

..todo:: Support is limited to just vms and nets for the time being due to a vm-only supported suffixes for *object_strs*.

parse_object_from_objects(*test_objects*, *param_dict=None*, *verbose=False*)
Parse a unique composite object from joined pre-parsed component objects.

Parameters

- **test_objects** – fully parsed test objects to parse the composite from
- **param_dict** (*{str, str} or None*) – runtime parameters used for extra customization
- **verbose** (*bool*) – whether to print extra messages or not

Type test_objects: (TestObject)

Returns parsed test objects

Return type [TestObject]

Raises exceptions.AssertionError if the parsed composite is not unique

parse_objects(*param_dict=None*, *object_strs=None*, *verbose=False*, *skip_nets=False*)
Parse all available test objects and their configurations or a selection of such where the selection is defined by the object string keys.

Parameters

- **param_dict** (*{str, str} or None*) – runtime parameters used for extra customization
- **object_strs** (*{str, str}*) – object-specific names and variant restrictions
- **verbose** (*bool*) – whether to print extra messages or not
- **skip_nets** (*bool*) – whether to skip parsing nets from current vms

Returns parsed test objects

Return type [TestObject]

parse_node_from_object(*test_object*, *param_dict=None*, *param_str=''*, *prefix=''*)
Get the original install test node for the given object.

Parameters

- **test_object** – fully parsed test object to parse the node from
- **param_dict** (*{str, str} or None*) – extra parameters to be used as overwrite dictionary
- **param_str** (*str*) – string block of parameters to be used as overwrite string
- **prefix** (*str*) – extra name identifier for the test to be run

Type test_object: NetObject

Returns parsed test node for the object

Return type TestNode

Raises AssertionError if the node is parsed from a non-net object

parse_nodes(*test_graph*, *param_dict=None*, *nodes_str=''*, *prefix=''*, *verbose=False*)
Parse all user defined tests (leaf nodes) using the nodes restriction string and possibly restricting to a single test object for the singleton tests.

Parameters

- **test_graph** (TestGraph) – test graph of already parsed test objects used to also validate test object uniqueness and main test object
- **param_dict** (*{str, str} or None*) – runtime parameters used for extra customization
- **nodes_str** (*str*) – block of node-specific variant restrictions
- **prefix** (*str*) – extra name identifier for the test to be run
- **verbose** (*bool*) – whether to print extra messages or not

Returns parsed test nodes

Return type [TestNode]

Raises param.EmptyCartesianProduct if no result on preselected vm

parse_object_nodes(*param_dict=None*, *nodes_str=''*, *object_strs=None*, *prefix=''*, *verbose=False*)
Parse test nodes based on a selection of parsable objects.

Returns parsed test nodes and test objects

Return type ([TestNode], [TestObject])

Raises param.EmptyCartesianProduct if no test variants for the given vm variants

The rest of the parameters are identical to the methods before.

We will parse all available objects in the configs, then parse all selected nodes and finally restrict to the selected objects specified via the object strings (if set) on a test by test basis.

parse_object_trees(*param_dict=None*, *nodes_str=''*, *object_strs=None*, *prefix=''*, *verbose=False*)
Parse all user defined tests (leaves) and their dependencies (internal nodes) connecting them according to the required/provided setup states of each test object (vm) and the required/provided objects per test node (test).

Returns parsed graph of test nodes and test objects

Return type TestGraph

The rest of the parameters are identical to the methods before.

The parsed structure can also be viewed as a directed graph of all runnable tests each with connections to its dependencies (parents) and dependables (children).

resolve(*reference*)
Discover (possible) tests from test references.

Parameters **reference** (*str or None*) – tests reference used to produce tests

Returns test factories as tuples of the test class and its parameters

Return type [(type, {str, str})]

__abstractmethods__ = frozenset()

1.2.4 avocado_i2n.params_parser module

1.2.4.1 SUMMARY

Module for handling all Cartesian config parsing and making it reusable and maximally performant.

Copyright: Intra2net AG

1.2.4.2 INTERFACE

exception avocado_i2n.params_parser.**EmptyCartesianProduct**(*message*)
Bases: Exception

Empty Cartesian product of variants

__init__(*message*)
Initialize an empty Cartesian product exception.

Parameters **message** (*str*) – additional message about the exception

avocado_i2n.params_parser.**custom_configs_dir**()
Custom directory for all config files.

avocado_i2n.params_parser.**tests_ovrwrt_file**()
Overwrite config file for all tests (nodes).

avocado_i2n.params_parser.**vms_ovrwrt_file**()
Overwrite config file for all vms (objects).

class avocado_i2n.params_parser.**ParsedContent**(*content*)
Bases: object

Class for parsed content of a general type.

__init__(*content*)
Initialize the parsed content.

reportable_form()
Parsed content representation used in reports of parsing steps.

Returns resulting report-compatible string

Return type str

Raises NotImlementedError as this is an abstract method

parsable_form()
Convert parameter content into parsable string.

Returns resulting parsable string

Return type str

Raises NotImlementedError as this is an abstract method

class avocado_i2n.params_parser.**ParsedFile**(*content*)
Bases: *avocado_i2n.params_parser.ParsedContent*

Class for parsed content of file type.

__init__(*content*)
Initialize the parsed content.

reportable_form()
Parsed file representation used in reports of parsing steps.

Arguments are identical to the ones of the parent class.

parsable_form()
Convert parameter file name into parsable string.

Returns resulting parsable string

Return type str

class avocado_i2n.params_parser.**ParsedStr**(*content*)
Bases: *avocado_i2n.params_parser.ParsedContent*

Class for parsed content of string type.

reportable_form()
Parsed string representation used in reports of parsing steps.

Arguments are identical to the ones of the parent class.

parsable_form()
Convert parameter string into parsable string.

Returns resulting parsable string

Return type str

This is equivalent to the string since the string is parsable by definition.

class avocado_i2n.params_parser.**ParsedDict**(*content*)
Bases: *avocado_i2n.params_parser.ParsedContent*

Class for parsed content of dictionary type.

reportable_form()
Parsed dictionary representation used in reports of parsing steps.

Arguments are identical to the ones of the parent class.

parsable_form()
Convert parameter dictionary into parsable string.

Returns resulting parsable string

Return type str

class avocado_i2n.params_parser.**Reparsable**
Bases: object

Class to represent quickly parsable Cartesian configuration, producing both parser and parameters (parser dicts) on demand.

__init__()
Initialize the parsable structure.

parse_next_file(*pfile*)
Add a file parsing step.

Parameters **pfile** (*str*) – file to be parsed next

If the parsable file has a relative form (not and absolute path), it will be searched in the relative test suite config directory.

parse_next_str(*pstring*)
Add a string parsing step.

Parameters **pstring** (*str*) – string to be parsed next

parse_next_dict(*pdict*)
Add a dictionary parsing step.

Parameters **pdict** (*{str, str}*) – dictionary to be parsed next

parse_next_batch(*base_file=None*, *base_str=''*, *base_dict=None*, *ovrwrt_file=None*, *ovrwrt_str=''*, *ovrwrt_dict=None*)
Parse a batch of base file, string, and dictionary, and possibly an overwrite file (with custom parameters at the user's home location).

Parameters

- **base_file** (*str or None*) – file to be parsed first
- **base_str** (*str or None*) – string to be parsed first
- **base_dict** (*{str, str} or None*) – params to be added first
- **ovrwrt_file** (*str or None*) – file to be parsed last
- **ovrwrt_str** (*str or None*) – string to be parsed last
- **ovrwrt_dict** (*{str, str} or None*) – params to be added last

The priority of the setting follows the order of the arguments: Dictionary with some parameters is topmost, string with some parameters is next and the file with parameters is taken as a base. The overwriting version is taken last, the base version first.

print_parsed()
Return printable information about what was parsed so far.

Returns structured text of the base/ovrwrt file/str/dict parse steps

Return type str

avocado_i2n.params_parser.**all_restrictions**()
Return all restrictions that can be passed for any test configuration.

Returns all available (from configuration) vms

Return type [str]

avocado_i2n.params_parser.**all_objects**(*key='vms'*, *composites=None*)
Return all test objects that can be passed for any test configuration.

Param str key: key to extract parametric objects from

Parameters **composites** (*[str]*) – composite restriction of the returned objects

Returns all available (from configuration) objects of a given type

Return type [str]

avocado_i2n.params_parser.**main_vm**()
Return the default main vm that can be passed for any test configuration.

Returns main available (from configuration) vm

Return type str or None

avocado_i2n.params_parser.**re_str**(*variant_str*, *base_str=''*, *tag=''*)
Add a variant restriction to the base string, optionally adding a custom tag as well.

Parameters

- **variant_str** (*str*) – variant restriction
- **base_str** (*str*) – string where the variant restriction will be added
- **tag** (*str*) – additional tag to the variant combination

Returns restricted parameter string

Return type str

avocado_i2n.params_parser.**join_str**(*variant_strs*, *base_str=''*)
Join all object variant restrictions over the base string.

Parameters

- **variant_strs** (*{str, str}*) – variant restrictions for each object as key, value pair
- **base_str** (*str*) – string where the variant restriction will be added

Returns restricted parameter string

Return type str

1.2.5 avocado_i2n.runner module

1.2.5.1 SUMMARY

Specialized test runner for the plugin.

Copyright: Intra2net AG

1.2.5.2 INTERFACE

class avocado_i2n.runner.**CartesianRunner**
Bases: avocado.core.plugin_interfaces.Runner

Test runner for Cartesian graph traversal.

description = 'Runs tests through a Cartesian graph traversal'

all_tests_ok
Evaluate if all tests run under this runner have an ok status.

Returns whether all tests ended with acceptable status

Return type bool

__init__()
Construct minimal attributes for the Cartesian runner.

name = 'traverser'

run_suite(*job*, *test_suite*)
Run one or more tests and report with test result.

Parameters

- **job** – job that includes the test suite
- **test_suite** (avocado.core.suite.TestSuite) – test suite with some tests to run

Returns a set with types of test failures

Return type set

__abstractmethods__ = frozenset()

run_terminal_node(*graph*, *object_name*, *params*, *slot*)
Run the set of tests necessary for creating a given test object.

Parameters

- **graph** (TestGraph) – test graph to run create node from
- **object_name** (*str*) – name of the test object to be created
- **params** (*{str, str}*) – runtime parameters used for extra customization

Raises NotImplementedError if using incompatible installation variant

The current implementation with implicit knowledge on the types of test objects internal spawns an original (otherwise unmodified) install test.

run_test(*job*, *node*)
Run a test instance inside a subprocess.

Parameters

- **job** (avocado.core.job.Job) – job that includes the test suite
- **node** (TestNode) – test node to run

run_test_node(*node*)

Run a node once, and optionally re-run it depending on the parameters.

Parameters **node** (TestNode) – test node to run

Returns run status of *run_test()*

Return type bool

Raises AssertionError if the ran test node contains no objects

The retry parameters are *retry_attempts* and *retry_stop*. The first is the maximum number of retries, and the second indicates when to stop retrying in terms of encountered test status and can be a list of statuses to stop on.

Only tests with the status of pass, warning, error or failure will be retried. Other statuses will be ignored and the test will run only once.

This method also works as a convenience wrapper around *run_test()*, providing some default arguments.

run_traversal(*graph*, *params*, *slot*)

Run all user and system defined tests optimizing the setup reuse and minimizing the repetition of demanded tests.

Parameters

- **graph** (TestGraph) – test graph to traverse
- **params** (*{str, str}*) – runtime parameters used for extra customization
- **slot** (*str*) – id name for the worker traversing the graph

Raises AssertionError if some traversal assertions are violated

The highest priority is at the setup tests (parents) since the test cannot be run without the required setup, then the current test, then a single child of its children (DFS), and finally the other children (tests that can benefit from the fact that this test/setup was done) followed by the other siblings (tests benefiting from its parent/setup.

Of course all possible children are restricted by the user-defined "only" and the number of internal test nodes is minimized for achieving this goal.

1.3 Module contents

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

G

H

I

Q

R

S

T

U

V