xrootd-tests Documentation

Release 0.1

CERN, Justin Lewis Salmon

October 23, 2012

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 General Overview

The XrdTest Framework is comprised of 3 main components:

- *Master*
- *Hypervisor*
- *Slave*

Each of which is explained in more detail below.

1.1.1 Master

Module file: `XrdTestMaster.py`

The master is the user entry point for the testing framework. The service is configured via a *ini*-style configuration file (see *Master configuration*).

It includes a web interface showing the current statistics of the service, as well as the status of the tests that are being run and have been run in the past (see *Using the XrdTest Web Interface*).

It accepts connections from slave and hypervisor daemons and dispatches commands to them. The master is responsible for running, orchestrating and synchronizing test suites.

Quick summary:

- User entry point to the framework
- Supervises and synchronizes all system activities
- Accepts connections from slaves and hypervisors and dispatches commands to them
- Runs as a system service (daemon), configured via batch of configuration files

1.1.2 Hypervisor

Module file: `XrdTestHypervisor.py`

The hypervisor receives cluster configurations from the master and starts/stops /configures the virtual machines which make up the cluster accordingly, including configuring the virtual network with which the slaves use to communicate. It uses `qemu` with the `kvm` kernel module in Linux and the `libvirt` virtualization API as a layer to communicate with `qemu`.

Quick summary:

- Component to manage clusters of virtual machines on demand of the master
- It is run as a system service (daemon), configured via configuration file (see *Hypervisor configuration*)
- Starts/stops/configures virtual machines
- Uses `libvirt` for managing virtual machines

1.1.3 Slave

Module file: `XrdTestSlave.py`

The slave component is installed on virtual or physical machines, and runs the actual tests. In the first iteration it will receive a bunch of shell scripts from the master and run them. Slaves connect to the master automatically, made possible by libvirt's use of dnsmasq.

Quick summary:

- The component which actually runs tests
- May be running on virtual or physical machines
- Runs as a system service (daemon), configured via configuration file (see *Slave configuration*)
- Receives test cases from the master and runs them synchronously with other slaves

1.1.4 Running as a system service (daemon)

If the application was installed from an RPM, it is automatically added to the system services (via `chkconfig`), thus it will be started automatically during system boot. It can also be started manually from the command line as follows:

```
service COMPONENT_NAME start
```

where COMPONENT_NAME is accordingly:

- `xrdtest-master`
- `xrdtest-slave`
- `xrdtest-hypervisor`

1.1.5 Running in debug mode

Note: The default location for configuration files is `/etc/XrdTest/<CONFIG_FILE_NAME>.conf`.

To start each component (master, hypervisor or slave) in debug mode, run the shell command below:

```
python /usr/sbin/XrdTestMaster.py -c XrdTestMaster.conf
```

One can start a hypervisor or a slave by replacing `XrdTestMaster` with `XrdTestHypervisor` or `XrdTestSlave` accordingly.

When running in debug mode, the component will print log messages to `stdout`, rather than writing them to the log file.

1.1.6 Running in background mode

To start a component in background mode (as a daemon), add the `-b` option to the shell command. It will then store log and PID files in their proper directories, as specified in the configuration file. For example:

```
# python /usr/sbin/XrdTestMaster.py -d -c XrdTestMaster.conf
```

1.2 Installation

The framework is available in RPM packages for Linux SLC6. Each application requires at least Python `2.4`. It comprises the following four main packages (the libraries required by them are also listed below):

- **xrdtest-lib**

  Dependencies: None

- **xrdtest-master**

  Dependencies:

  - `python-apscheduler-2.0.3`
  - `python-cheetah-2.4.1`
  - `python-cherrypy-3.1.2`
  - `python-inotify-0.9.1`
  - `python-uuid-1.30`
  - `pyOpenSSL-0.10-2`
  - `python-ssl-1.15`

- **xrdtest-hypervisor**

  Dependencies:

  - `python-ssl-1.15`
  - `libvirt-python-0.9.3`
  - `libvirt-0.9.10`

- **xrdtest-slave**

  Dependencies:

  - `python-ssl-1.15`

To install each component, you must follow typical RPM installation instructions.

1.3 Master configuration

This section describes how to configure the XrdTest Master framework component, including how to set up a repository to hold test suite and cluster definitions, web interface options, logging options and security configurations.

Note: The default location for configuration files is `/etc/XrdTest/<CONFIG_FILE_NAME>.conf`.

The master configuration file uses the *ini*-style format of the python `ConfigParser` module. There are multiple sections, each of which is explained separately below. First, the configuration directive will be given, followed by an explanation.

1.3.1 Configuration sections

`[general]`

```
test-repos=remote,local
```

A list of repositories to use, each of which must have a corresponding [test-repo-<reponame>] section below. As an example, we use two test suites: one local (`test-repo-local`), and one in a remote `git` repository (`test-repo-remote`).

```
suite_sessions_file=/var/log/XrdTest/suite_history.bin
```

The path to the file which stores previous test suite history.

`[test-repo-remote]`

The section for the first of our two example repositories. This repository is a remote `git` repository. Currently, the framework supports localfs and `git` repositories only. It is planned to include `svn` support in the future.

Note: You need passwordless access to the repository for this to work (such as key-based SSH, Kerberos, or a HTTP URL). Password based authentication will not work, as synchronization of the remote repository happens automatically at certain time intervals.

```
# Example settings for a remote git repository.
type=git

# Path to the remote repository. Accepts any valid Git URL.
remote_repo=jsalmon@xrootd.cern.ch:/var/repos/xrootd-testsuite.git

# Which local/remote branches to use.
remote_branch=origin/master
local_branch=master

# Path where the remote repo will be checked out locally.
local_path=/var/tmp/xrootd-testsuite

# Paths to the local checkouts of cluster and test suite definitions.
```

```
cluster_defs_path=clusters
suite_defs_path=test-suites
```

Each directive should be fairly self-explanatory. The `remote_repo` directive **accepts any valid git URL**.

It is necessary to provide a path where the remote repository will be checked out, as the system in fact clones the remote repository to this local path, does `fetch`/`diff` periodically, then does `pull` if there are changes in the remote repo.

It is also necessary to point to the directories which hold cluster and test suite definitions **inside the local checkout directory**. This is in case you want to change the naming conventions to better suit your environment.

[test-repo-local]

The section for the second example repository. This repository is located in the local filesystem, and is much simpler to configure than a remote one.

```
# Example settings for a local repository of cluster/test suite definitions.
type=localfs

local_path=/var/repos/xrootd-testsuite
cluster_defs_path=clusters
suite_defs_path=test-suites
```

You need to point to the top directory, and the subdirectories which hold cluster and test suite definitions.

[server]

```
# Password to authenticate hypervisors.
connection_passwd=some_password

# The IP and port the master will listen on.
ip=0.0.0.0
port=20000
```

[webserver]

```
# Absolute path to webpage files (defaults to /usr/share/XrdTest/webpage).
# Uncomment and add your path to change the web root.
webpage_dir=/usr/share/XrdTest/webpage

# Protocol to use for the web server. Defaults to HTTP.
protocol=https

# The port to access the web interface on. Defaults to 8080 for HTTP and 8443
# for HTTPS.
port=8443

# The password that allows running test suites via the webpage (defaults to none)
# suite_run_pass=somepass
```

```
[scheduler]

# If set to 0, the scheduler will not run, strangely enough.
enabled=1


[security]

# Location of the master's SSL certificate and private key. Will be generated
# automatically at install time. Don't change these.
certfile=/etc/XrdTest/certs/mastercert.pem
keyfile=/etc/XrdTest/certs/masterkey.pem

# Location of the key/certificate which the master will use to become it's own
# CA (for signing CSRs from slaves which need to use GSI).
ca_certfile=/etc/XrdTest/certs/cacert.pem
ca_keyfile=/etc/XrdTest/certs/cakey.pem


[daemon]

# Path to PID file if being run in daemon mode.
pid_file_path=/var/run/XrdTestMaster.pid

# Path the the master's log file.
log_file_path=/var/log/XrdTest/XrdTestMaster.log

# Amount of information to log. Constants from standard python logging module.
# Defaults to INFO. Possible values: NOTSET (off), ERROR (only errors), WARN
# (warnings and above), INFO (most logs), DEBUG (everything)
log_level=DEBUG
```

1.3.2 Other considerations

- Firewall (tcp on port 10000)

1.4 Hypervisor configuration

1.4.1 Configuration sections

```
[test_master]

# IP and port of the XrdTest Master.
ip=somehost.somedomain.com
port=20000

# Password to authenticate with the master.
connection_passwd=some_passwd
```

```
[virtual_machines]

# Path to the KVM executable.
# emulator_path=/usr/bin/kvm
emulator_path=/usr/libexec/qemu-kvm

# Name of the libvirt storage pool in which slave boot images will be placed.
# You must configure this storage pool yourself, and place any boot images as
# libvirt storage volumes into the pool. This pool can be anywhere (NAS, NFS
# etc), as long as it is visible as a libvirt storage pool on this hypervisor.
storage_pool=XrdTest


[security]

# Paths to SSL certificates and keys for the hypervisor.
certfile=/etc/XrdTest/certs/hypervisorcert.pem
keyfile=/etc/XrdTest/certs/hypervisorkey.pem


[daemon]

# Path to the PID file for the hypervisor when running as daemon.
pid_file_path=/var/run/XrdTestHypervisor.pid

# Where the hypervisor writes its logs
log_file_path=/var/log/XrdTest/XrdTestHypervisor.log

# Amount of information to log. Constants from standard python logging module.
# Defaults to INFO. Possible values: NOTSET (off), ERROR (only errors), WARN
# (warnings and above), INFO (most logs), DEBUG (everything)
log_level=INFO
```

1.4.2 Other considerations

- Available memory, storage pool size

1.5 Slave configuration

1.5.1 Configuration sections

```
[test_master]

# IP and port of the XrdTest Master. Slaves can set this to master.xrd.test,
# as the virtual network will have a DNS entry which will resolve back to the
# actual master IP.
ip=master.xrd.test
port=20000
```

```
[security]

# Paths to SSL certificates and keys for the slave.
certfile=/etc/XrdTest/certs/slavecert.pem
keyfile=/etc/XrdTest/certs/slavekey.pem


[daemon]

# Path to the PID file for the slave when running as daemon.
pid_file_path=/var/run/XrdTestSlave.pid

# Where the slave writes its logs
log_file_path=/var/log/XrdTest/XrdTestSlave.log

# Amount of information to log. Constants from standard python logging module.
# Defaults to INFO. Possible values: NOTSET (off), ERROR (only errors), WARN
# (warnings and above), INFO (most logs), DEBUG (everything)
log_level=DEBUG
```

1.5.2 Other considerations

- Slave image config (network, size, OS, root password etc)

1.6 Writing test suites

This page describes how to write test suites for the XrdTest Framework. For details on how to set up a repository to hold your test suites, see *Master configuration*.

Note: Full examples can be found in the `examples` directory.

1.6.1 Structure of a test suite

Test suites have a specific structure which must be adhered to, explained below:

- Each test suite resides in its own directory.
- Each test suite has a definition file, which uses Python syntax. It is loaded dynamically as a Python function at runtime, so it must be syntactically correct. It must have a specific name (see *The definition file* below).
- Each test suite has a mandatory global **initialization** script, which is used to set up the (xrootd) environment ready for running test cases (see *Writing initialization/run/finalization scripts* below).
- Test suites can optionally have a global **finalization** script, generally used to perform cleanup tasks, such as removing files from data servers, removing authentication credentials etc. (see *Writing initialization/run/finalization scripts* below).
- Each test suite has a subdirectory called **tc** which holds the set of test cases for this suite.
  - Each test case resides in its own subdirectory. The name of the directory defines the name of the test case.

- Each test case has a mandatory **initialization** script
- Each test case has a mandatory **run** script.
- Test cases can optionally have a **finalization** script.

The definition file

The test suite definition file must be in the root directory of the test suite directory, and it must have the same name as the folder, with a `.py` extension.

A test suite is defined inside a function named `getTestSuite()` which takes no parameters. Here is an example of the beginning of a definition file:

```
from XrdTest.TestUtils import TestSuite

def getTestSuite():

    ts = TestSuite()
    ts.name = "ts_002_frm"
```

The `ts.name` attribute is the unique name of the test suite. It must match the name of the file exactly (minus the *.py* extension) and also match the directory name in which this test suite resides.

The test suite name is arbitrary, but in the CERN `xrootd-testsuite` repository we have a naming convention of `ts_<numerical id>_<shorthand description>`. For example, the suite which tests GSI functionality is named **ts_006_gsi**. You are of course free to choose your own naming conventions, however.

Defining required clusters

To define the cluster(s) which this test suite requires, include a line like this:

```
ts.clusters = ['cluster_002_frm']
```

This line is mandatory. Currently, there is only support for one cluster per test suite. It is planned to have the ability to run multiple clusters on multiple hypervisors in the future. For information on how to define a cluster, see *Writing cluster definitions*.

There is also the ability to specify a subset of the machines in a cluster, with a line like this:

```
ts.machines = ['frm1', 'frm2', 'ds1', 'ds2', 'client1']
```

There haven't been any use cases where this has been needed yet, but the functionality exists if one comes along. This line os not mandatory.

Scheduling test suites

To schedule the test suite to be run at particular intervals (cron-style), you must include a line like this:

```
ts.schedule = dict(second='30', minute='08', hour='*', day='*', month='1')
```

This line is mandatory.

Defining what is run

To define which test cases will be run in this suite, include a line similar to this:

```
ts.tests = ['copy_to', 'copy_from']
```

This line is mandatory. If a test case exists in the **tc** directory, but is not included in the line in the definition file, it will not be run.

To point to the suite initialization script, include a line like this:

```
ts.initialize = "file://suite_init.sh"
```

This line can be a relative file URL (as above), an absolute file URL, or a HTTP URL. The initialization script is mandatory.

To point to the suite finalization script, include a line like this:

```
ts.finalize = "file://suite_finalize.sh"
```

The finalization script is not mandatory. It can be used for general cleaning up after all test cases have been run.

Including log files

The framework has functionality for retrieving arbitrary log files from each slave at each stage of the test suite. To use this feature, include a line like this:

```
ts.logs = ['/var/log/xrootd/@slavename@/xrootd.log',
           '/var/log/xrootd/@slavename@/cmsd.log',
           '/var/log/XrdTest/XrdTestSlave.log']
```

You should provide the path to any log files which will be useful to inspect. It is possible to use the @slavename@ tag in the log file path (See *The @tag@ system* for an explanation of the @slavename@ and other tags). It can be useful to include the slave log (XrdTestSlave.log) for debugging purposes.

Getting email alerts

It is possible to give an arbitrary list of email addresses, each of which can be notified of the outcome of a test suite run, to a specified level of verbosity.

The list of email addresses is defined with a line like the following:

```
ts.alert_emails = ['jsalmon@cern.ch', 'foo@bar.com']
```

The amount of email alerts to be sent is configured with policy lines like the following:

```
ts.alert_success = 'SUITE'
ts.alert_failure = 'CASE'
```

There is a separate policy for failure notifications and success notifications for flexibility. The possible options for both policies are:

- `SUITE` - Send an email about the final state of the entire test suite (success or failure).

- `CASE` - Send an email about the final state of each individual test case (success or failure). Implied SUITE.
- `NONE` - Don't send any emails.

The default options are generally OK, i.e. `CASE` for failure alerts (as you want to know if the test suite failed and also which individual test cases failed) and `SUITE` for success alerts (you don't care if each test case succeeds, only that the whole suite succeeds). You might want to put `NONE` for the success policy if you really only care about failures.

1.6.2 Writing initialization/run/finalization scripts

As mentioned earlier, each test suite has a mandatory global initialization script, an optional global finalization script, and a set if initialization/ run/finalization scripts for each test case.

The framework has been designed in this way, so that actions can be synchronized between participants (slaves) in the cluster. For example, if a slave completes its global initialization script, it will wait for all other slaves to complete theirs before moving on to the next stage. Similarly, a slave will not begin the run stage of a test case until it and all other slaves have completed the test case initialization stage. The XrdTest Master is actually responsible for orchestrating this activity.

It is important to note that should the global initialization script fail on any slave for any reason, then the **entire test suite** will be considered as failed, and no test cases will be run. A command that returns a non-zero exit code is considered as a failure, unless specifically stated otherwise by using the `assert_fail` function (see *Available functions* below).

If a **test case** initialization script fails, the suite will continue to run. The same is true for the remaining stages of the suite.

Also note that you do not need to worry about *stdout* and *stderr*. Anything that is printed to *stderr* will be redirected to *stdout*. This is due to both ease of use, and to problems with Python's `subprocess` module and the way it handles *stderr*.

The framework provides some features to make the scripts more flexible,explained below.

The `@tag@` system

There are some special keywords which can be used inside any test suite script. These keywords, or *tags*, have a descriptive name enclosed with `@` symbols. Each tag within a script will be replaced with an appropriate real value at runtime, based upon which slave is currently running the script, the cluster configuration, and the parameters with which the master is to be contacted.

The currently available tags are as follows:

- `@slavename@` - The FQDN of the current slave running the script. This allows one to write a single script containing if/else blocks to determine which piece of code the current slave will run, based upon its name.
- `@port@` - The port on which the master should be contacted (defined in the master configuration file, see *Master configuration*).
- `@proto@` - The protocol by which the master should be contacted (defined in the master configuration file, see *Master configuration*).
- `@diskmounts@` - Gets resolved to the appropriate disk mount command(s) for the current slave. Disks are always mounted as `ext4` using `user_xattr`.

It is planned to allow user extensions to the tagging system sometime in the future, so that arbitrary tags can be used inside scripts for even greater flexibility.

Available functions

There is a small library of functions (located in */etc/XrdTest/utils*) that can be used by default in test scripts. To use these functions, simply source the file inside the script like this:

```
#!/bin/bash
source /etc/XrdTest/utils/functions.sh
```

A brief description of the currently available functions:

- `assert_fail` - a function to assert the non-zero exit code of a function. Used for testing invalid use cases and verifiying that they fail as they should. For example:

  ```
  assert_fail rm non_existent
  ```

  will return zero and not cause the script to exit (as would have happened if the `assert_fail` command were not used).

- `log` - Used for timestamping and printing single-line commands or progress messages. For example:

  ```
  log "Initializing test suite on slave @slavename@"
  ```

  will print a timestamped line in the session log which looks like this:

  ```
  [10:49:20] Initializing test suite on slave manager1.xrd.test
  ```

- `stamp` - Used for timestamping and printing entire command outputs. For example:

  ```
  stamp ls -al /data
  ```

  will produce output like this:

  ```
  [09:57:37]  total 51208
  [09:57:37]  drwxr-xr-x.  2 daemon daemon     4096 Oct 22 09:57 .
  [09:57:37]  dr-xr-xr-x. 25 root   root       4096 Oct 22 09:52 ..
  [09:57:37]  -rw-r--r--.  1 root   root   52428800 Oct 22 09:57 some_file
  ```

1.7 Writing cluster definitions

```
from XrdTest.ClusterUtils import Cluster, Network, Host, Disk

def getCluster():
    cluster = Cluster()
    #--------------------------------------------------------------------------
    # Global names
    #--------------------------------------------------------------------------
    cluster.name = 'cluster_example'
    network_name = cluster.name + '_net'

    #--------------------------------------------------------------------------
    # Cluster defaults
```

```
#
# The bootImage parameter is relative to some libvirt-managed storage pool.
#-------------------------------------------------------------------------
cluster.defaultHost.bootImage = 'slc6_testslave_ref.img'
cluster.defaultHost.cacheBootImage = True
cluster.defaultHost.arch = 'x86_64'
cluster.defaultHost.ramSize = '1048576'
cluster.defaultHost.net = network_name

#-------------------------------------------------------------------------
# Host definitions
#-------------------------------------------------------------------------
metamanager1 = Host('metamanager1.xrd.test', '192.168.127.3', "52:54:00:65:44:65")
manager1 = Host('manager1.xrd.test', '192.168.127.4', "52:54:00:65:44:66")
manager2 = Host('manager2.xrd.test', '192.168.127.5', "52:54:00:65:44:67")
ds1 = Host('ds1.xrd.test', '192.168.127.6', "52:54:00:65:44:68")
ds2 = Host('ds2.xrd.test', '192.168.127.7', "52:54:00:65:44:69")
ds3 = Host('ds3.xrd.test', '192.168.127.8', "52:54:00:65:44:70")
ds4 = Host('ds4.xrd.test', '192.168.127.9', "52:54:00:65:44:71")
client1 = Host('client1.xrd.test', '192.168.127.10', "52:54:00:65:44:72")

#-------------------------------------------------------------------------
# Additional host disk definitions
#
# As per the libvirt docs, the device name given here is not guaranteed to
# map to the same name in the guest OS. Incrementing the device name works
# (i.e. disk1 = vda, disk2 = vdb etc.).
#
# Disk sizes should be larger than 10GB for data server nodes, otherwise
# the node might not be selected by the cmsd.
#-------------------------------------------------------------------------
metamanager1.disks =  [Disk('disk1', '20G', device='vda', mountPoint='/data')]
manager1.disks =  [Disk('disk1', '20G', device='vda', mountPoint='/data')]
manager2.disks =  [Disk('disk1', '20G', device='vda', mountPoint='/data')]
ds1.disks =  [Disk('disk1', '20G', device='vda', mountPoint='/data')]
ds2.disks =  [Disk('disk1', '20G', device='vda', mountPoint='/data')]
ds3.disks =  [Disk('disk1', '20G', device='vda', mountPoint='/data')]
ds4.disks =  [Disk('disk1', '20G', device='vda', mountPoint='/data')]
client1.disks =  [Disk('disk1', '20G', device='vda', mountPoint='/data')]

#-------------------------------------------------------------------------
# Network definition
#-------------------------------------------------------------------------
net = Network()
net.bridgeName = 'virbr_example'
net.name = network_name
net.ip = '192.168.127.1'
net.netmask = '255.255.255.0'
net.DHCPRange = ('192.168.127.2', '192.168.127.254')
#-------------------------------------------------------------------------
# Optional load balancing configuration
#-------------------------------------------------------------------------
# The DNS alias to be used
net.lbAlias = 'lb.xrd.test'
# The machines that will be load balanced (round-robin) under the alias
net.lbHosts = [ds1, ds2, ds3, ds4]
```

```
    # Hosts to be included in the cluster
    hosts = [metamanager1, manager1, manager2, ds1, ds2, ds3, ds4, client1]

    net.addHosts(hosts)
    cluster.network = net
    cluster.addHosts(hosts)
    return cluster
```

1.8 Using the XrdTest Web Interface

- Available method calls
- Running test suites manually (password)
- Activating test suites
- CA
- SSS distribution

1.9 Source code reference manual

1.9.1 XrdTest Package

XrdTest Package

Library files for the XrdTest Framework

ClusterManager Module

class XrdTest.ClusterManager.**ClusterManager**
Virtual machines clusters' manager

attachDisk(*host*, *diskName*, *diskSize*, *cache*, *device*)
TODO:

attachDisks(*host*)

copyImg(*huName*, *safeCounter=None*)
Method runnable in separate threads, to separate copying of source operating system image to a temporary image.

@param huName: host.uname - host unique name @param safeCounter: thread safe counter to signalize this run finished

createCluster(*cluster*)
Creates cluster: first network, then virtual machines - hosts. If it get request to create machine (host) that already exists (with the same name) - it removes it completely. The same story regards network. @param cluster: cluster definition object

createNetwork(*networkObj*, *clusterName*)
Creates and starts cluster's network. It utilizes defineNetwork at first and doesn't create

network definition if it already exists. @param networkObj: Network object @raise ClusterManagerException: when fails @return: None

defineHost (*host*)
Defines virtual host in a cluster using given host object, not starting it. Host with the given name may be defined once in the system. Stores hosts objects in class property self.hosts.

@param host: ClusterManager.Host object @raise ClusterManagerException: when fails @return: host object from libvirt lib

defineNetwork (*netObj*)
Defines network object without starting it. @param xml: libvirt XML cluster definition @raise ClusterManagerException: when fails @return: None

disconnect ()
Undefines and removes all virtual machines and networks created by this cluster manager and disconnects from libvirt manager. @raise ClusterManagerException: when fails

findStoragePool (*poolname*)
Attempt to find a storage pool with the given name.

findStorageVolume (*poolname*, *volumename*)
Attempt to find a storage volume (file) in the specified libvirt storage pool. If the volume is not found, the default pool will be searched. Return the full path to the volume.

getPoolPath (*poolXML*)
Parse the given storage pool XML description and return its path.

removeCluster (*clusterName*)

removeDanglingHost (*hostObj*)
Remove already defined host, if it has name the same as hostObj. @param hostObj:

removeDanglingNetwork (*netObj*)
Remove already defined network, if it has name the same as netObj. @param hostObj:

removeHost (*hostUName*)
Can not be used inside loop iterating over hosts! @param hostUName: host.uname host unique name

removeHosts (*hostsUnameList*)
Remove multiple hosts. @param hostsUnameList: list of unames of defined hosts.

removeNetwork (*netUName*)
Can not be used inside loop iterating over networks! @param hostName:

updateState (*state*, *clusterName*)
Send a progress update message to the master.

virtconnect (*url='qemu:///system'*)
Creates and returns connection to virtual machines manager @param url: connection url @raise ClusterManagerException: when fails to connect @return: None

ClusterUtils Module

class XrdTest.ClusterUtils.**Cluster**
Bases: XrdTest.Utils.Stateful

S_ACTIVE = (8, 'Cluster active.')

S_ATTACHING_DISKS = (6, 'Attaching slave disks.')

S_COPYING_IMAGES = (5, 'Copying slave images.')

S_CREATING_NETWORK = (3, 'Creating network.')

S_CREATING_SLAVES = (4, 'Creating slaves.')

S_DEFINED = (0, 'Cluster defined correctly.')

S_DEFINITION_SENT = (1, 'Cluster start command sent to hypervisor.')

S_DESTROYING_CLUSTER = (10, 'Destroying cluster')

S_ERROR = (-4, 'Cluster error')

S_ERROR_START = (-3, 'Error at start:')

S_ERROR_STOP = (-2, 'Error at stop:')

S_STARTING_CLUSTER = (2, 'Starting cluster.')

S_STOPCOMMAND_SENT = (9, 'Cluster stop command sent to hypervisor.')

S_STOPPED = (11, 'Cluster stopped.')
Represents a cluster comprised of hosts connected through network.

S_UNKNOWN = (0, 'Cluster state unknown.')

S_UNKNOWN_NOHYPERV = (-1, 'Cluster state unknown: no hypervisor to run cluster.')

S_WAITING_SLAVES = (7, 'Waiting for slaves to connect.')

addHost(*host*)

addHosts(*hosts*)

network

networkGet()

networkSet(*net*)

setEmulatorPath(*emulator_path*)

validateAgainstSystem(*clusters*)
Check if cluster definition is correct with other clusters defined in the system. This correctness is critical for cluster definition to be added. @param clusters:

validateDynamic()
Check if Cluster definition is semantically correct i.e. on the hypervisor's machine e.g. if disk images really exists on the machine it's to be planted.

validateStatic()
Check if Cluster definition is correct and sufficient to create a cluster.

exception XrdTest.ClusterUtils.**ClusterManagerException**(*desc*, *typeFlag=1*)
Bases: exceptions.Exception

General Exception raised by module

class XrdTest.ClusterUtils.**Disk**(*name*, *size*, *device='vda'*, *mountPoint='/data'*, *cache=True*)
Bases: object

parseDiskSize(*size*)
Takes a size in readable format, e.g. 50G or 200M and returns the size in bytes.

If a purely numeric string is given, a byte value will be assumed.

class XrdTest.ClusterUtils.**Host**(*name=''*, *ip=''*, *net=''*, *ramSize=''*, *arch=''*, *bootImage=None*, *cacheBootImage=True*, *emulatorPath=''*, *uuid=''*)
Bases: object

Represents a virtual host which may be added to network

randMac()

uname
Return unique name of the machine within cluster's namespace.

xmlDesc

xmlDomainPattern = "\n<domain type='kvm'>\n <name>%(uname)s</name>\n <uuid>%(uuid)s</uuid

class XrdTest.ClusterUtils.**Network**
Bases: object

Represents a virtual network

addDHCPHost(*host*)

addDnsHost(*host*)

addHost(*host*)
Add host to network. First to DHCP and then to DNS. @param host: tuple (MAC address, IP address, HOST fqdn)

addHosts(*hostsList*)
Add hosts to network. @param param: hostsList

uname
Return unique name of the machine within cluster's namespace.

xmlDesc

xmlDescPattern = '\n<network>\n <name>%(name)s</name>\n <dns>\n <txt name="xrd.test" value="

xmlDnsHostPattern = '\n <host ip="%(ip)s">\n %(lbalias)s\n <hostname>%(hostname)s</hostname>\n

xmlHostPattern = '\n <host mac="%(mac)s" name="%(name)s" ip="%(ip)s" />\n'

XrdTest.ClusterUtils.**extractClusterName**(*path*)

XrdTest.ClusterUtils.**getFileContent**(*filePath*)
Read and return whole file content as a string @param filePath:

XrdTest.ClusterUtils.**loadClusterDef**(*fp*, *clusters*, *validateWithRest=True*)

XrdTest.ClusterUtils.**loadClustersDefs**(*path*)
Loads cluster definitions from .py files stored in path directory @param path: path for .py files, storing cluster definitions

Daemon Module

class XrdTest.Daemon.**Daemon**(*progName*, *pidFile*, *logFile*)

Represents and manages running daemon of a given runnable object. For initialization it requires object that inherits class Runnable.

check(*pid=None*)

Checks if process with given pid is currently running. If no pid is given, it tries to retrieve pid from the pidFile given in the constructor of this class.

@param pid: pid of process to be checked @return: pid (if process runs), None (otherwise)

reload(*pid=None*)

Reloads the daemon by sending SIGHUM

removePidFile()

Remove pid file if it exists

start(*runnable*)

Starts the daemon as a separate process.

@param runnable: instance of runnable object (inherits Runnable).

stop(*pid=None*)

Stop the deamon

exception XrdTest.Daemon.**DaemonException**(*desc*)

Bases: exceptions.Exception

General Exception raised by Daemon.

class XrdTest.Daemon.**Runnable**

Bases: object

Abstract basic class for object to be runned as a daemon. @note: children class it should handle SIGUP signal or it will suspend

run()

Main jobs of programme. Has to be implemented.

DirectoryWatch Module

class XrdTest.DirectoryWatch.**ClustersDefinitionsChangeHandler**(*pevent=None*, ***kwargs*)

Bases: pyinotify.ProcessEvent

If cluster definition file changes - it runs.

process_default(*event*)

Actual method that handle incoming dir change event. @param event:

class XrdTest.DirectoryWatch.**DirectoryWatch**(*repo*, *config*, *callback*, *watch_type=None*)

Bases: object

Base class for monitoring directories and invoking callback on change. Instantiation of this class defines which type of watch will happen (local or remote)

IN_CREATE = 256L

IN_DELETE = 512L

IN_MODIFY = 2L

IN_MOVED = 192L

mask = 962L

watch()

watch_localfs()
Monitor a local directory for changes.

watch_remote_git()
Monitor a remote git repository by polling at a set interval.

class XrdTest.DirectoryWatch.**SuiteDefinitionsChangeHandler**(*pevent=None*, ***kwargs*)

Bases: pyinotify.ProcessEvent

If suite definition file changes it runs

process_default(*event*)
Actual method that handle incoming dir change event. @param event:

GitUtils Module

XrdTest.GitUtils.**git_clone**(*remote_repo*, *local_repo*, *cwd*)
Clone a remote repository into a new local directory. Must have key-based authentication set up for this to work.

TODO: handle exceptions for no key-based auth

@param remote_repo: the repository repo on the remote host. @param local_repo: the local repo in which to clone the new repo. @param cwd: the working directory in which to execute.

XrdTest.GitUtils.**git_diff**(*local_branch*, *remote_branch*, *cwd*)
Perform a diff operation between a local and remote repository.

@param local_branch: the local repository branch name. @param remote_branch: the remote branch name. @param cwd: the working directory in which to execute.

XrdTest.GitUtils.**git_fetch**(*cwd*)
Fetch objects and refs from a remote repository.

@param cwd: the working directory in which to execute.

XrdTest.GitUtils.**git_pull**(*cwd*)
Fetch from and merge with a remote repository.

@param cwd: the working directory in which to execute.

XrdTest.GitUtils.**sync_remote_git**(*repo*, *config*)
Fetch the status of a remote git repository for new commits. If new commits, pull the new changes.

Need key-based SSH authentication for this method to work. Also, on AFS systems like lxplus, a valid kerberos ticket is needed.

@param repo: @param config: configuration file containing repository information

Job Module

class XrdTest.Job.**Job**(*job*, *groupId=''*, *args=None*)
Bases: object

Keeps information about job, that is to be run. It's enqueued by scheduler and dequeued if fore coming job was handled.

FINALIZE_TEST_CASE = 5

FINALIZE_TEST_SUITE = 2

INITIALIZE_TEST_CASE = 3

INITIALIZE_TEST_SUITE = 1

RUN_TEST_CASE = 4

START_CLUSTER = 6

STOP_CLUSTER = 7

S_ADDED = (0, 'Job added to jobs list.')

S_STARTED = (1, 'Job started. In progress.')

TEST_JOB = 0

static genJobGroupId(*suite_name*)
Utility function to create unique name for group of jobs. @param suite_name:

SocketUtils Module

class XrdTest.SocketUtils.**FixedSockStream**(*sock*)
Bases: object

Wrapper for socket to ensure correct behaviour of send and recv.

close()

recv(*recvRaw=False*)

recvBounded(*toRecvLen*)

send(*obj*, *sendRaw=False*)

sendBounded(*msg*, *toSendLen*)

class XrdTest.SocketUtils.**PriorityBlockingQueue**
Bases: object

Synchronized priority queue. Pattern for entries is a tuple in the form: (priority_number, data). Lowest valued entries are retrieved first.

get()
Retrieves data of an element from (priority, data) with the lowest priority from the queue.

put(*elem*)
Puts element to the queue. @param elem: a tuple in the form: (priority_number[int], data).

rawGet()
Retrieves tuple element (priority, data) with the lowest priority from the queue.

exception XrdTest.SocketUtils.**SocketDisconnectedError**(*desc*)
Bases: exceptions.Exception

TODO

class XrdTest.SocketUtils.**XrdMessage**(*name*, *msg_sender=None*)
Bases: object

Network message passed between Xrd Testing Framework nodes.

M_CLUSTER_STATE = 'cluster_state'

M_DISCONNECT = 'disconnect'

M_HELLO = 'hello'

M_HYPERVISOR_STATE = 'hypervisor_state'

M_START_CLUSTER = 'start_cluster'

M_STOP_CLUSTER = 'stop_cluster'

M_TAG_REPLY = 'tag_reply'

M_TAG_REQUEST = 'tag_request'

M_TESTCASE_FINALIZE = 'test_case_finalize'

M_TESTCASE_INIT = 'test_case_init'

M_TESTCASE_RUN = 'test_case_run'

M_TESTCASE_STAGE_RESULT = 'test_case_stage_result'

M_TESTSUITE_FINALIZE = 'test_suite_finalize'

M_TESTSUITE_INIT = 'test_suite_init'

M_TESTSUITE_STATE = 'test_case_state'

M_UNKNOWN = 'unknown'

name = 'unknown'

sender = None

TCPClient Module

class XrdTest.TCPClient.**Hypervisor**(*socket*, *hostname*, *address*, *state*)
Bases: XrdTest.TCPClient.TCPClient

Wrapper for any hypervisor connection established.

class XrdTest.TCPClient.**Slave**(*socket*, *hostname*, *address*, *state*)
Bases: XrdTest.TCPClient.TCPClient

Wrapper for any slave connection established.

S_SUITE_FINALIZE_SENT = (12, 'Test suite finalize sent to slave')

S_SUITE_INITIALIZED = (11, 'Test suite initialized')

S_SUITE_INIT_SENT = (10, 'Test suite init sent to slave')

S_TEST_FINALIZED = (26, 'Test case finalized')

S_TEST_FINALIZE_SENT = (25, 'Sent test case finalize to slave')

S_TEST_INITIALIZED = (22, 'Test case initialized')

S_TEST_INIT_SENT = (21, 'Sent test case init to slave')

S_TEST_RUN_FINISHED = (24, 'Test case run finished')

S_TEST_RUN_SENT = (23, 'Sent test case run to slave')

class XrdTest.TCPClient.**TCPClient**(*socket*, *hostname*, *address*, *state*)
Bases: XrdTest.Utils.Stateful

Represents any type of TCP client that connects to XrdTestMaster. Base class for Hypervisor and Slave.

S_CONNECTED_IDLE = (1, 'Connected')

S_NOT_CONNECTED = (2, 'Not connected')

send(*msg*)

class XrdTest.TCPClient.**TCPReceiveThread**(*sock*, *recvQueue*)
Bases: object

TODO:

close()
TODO:

run()
TODO:

TCPServer Module

class XrdTest.TCPServer.**MasterEvent**(*e_type*, *e_data*, *msg_sender_addr=None*)
Bases: object

Wrapper for all events that comes to XrdTestMaster. MasterEvent can be message from slave or hypervisor, system event like socket disconnection, cluster or test suite definition file change or scheduler job initiation. It has priorities. PRIO_IMPORTANT is processed before PRIO_NORMAL.

M_CLIENT_CONNECTED = 2

M_CLIENT_DISCONNECTED = 3

M_HYPERV_MSG = 4

M_JOB_ENQUEUE = 6

M_RELOAD_CLUSTER_DEF = 7

M_RELOAD_SUITE_DEF = 8

M_SLAVE_MSG = 5

M_UNKNOWN = 1

PRIO_IMPORTANT = 1

PRIO_NORMAL = 9

class XrdTest.TCPServer.**ThreadedTCPRequestHandler**(*request*, *client_address*, *server*)

Bases: SocketServer.BaseRequestHandler

Client's TCP request handler.

C_HYPERV = 'hypervisor'

C_SLAVE = 'slave'

authClient(*clientType*)
Check if hypervisor is authentic. It will provide the connection password.

handle()
Handle new incoming connection and keep it to receive messages.

setup()
Initiate class properties

class XrdTest.TCPServer.**ThreadedTCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

Bases: SocketServer.ThreadingMixIn, XrdTest.TCPServer.XrdTCPServer

Wrapper to create threaded TCP Server.

class XrdTest.TCPServer.**XrdTCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

Bases: SocketServer.TCPServer

Wrapper for SocketServer.TCPServer, to enable setting beneath params.

allow_reuse_address = True

TestUtils Module

class XrdTest.TestUtils.**TestCase**
Represents a single test case object.

validateStatic()
Return whether or not definition (e.g given names) is statically correct.

class XrdTest.TestUtils.**TestSuite**
Bases: XrdTest.Utils.Stateful

Represents a single test suite object.

S_ALL_FINALIZED = (32, 'All machines finalized')

S_ALL_INITIALIZED = (22, 'All machines initialized')

S_ALL_TEST_FINALIZED = (48, 'Test case finalized on all slaves')

S_ALL_TEST_INITIALIZED = (42, 'Test case initialized on all slaves')

S_ALL_TEST_RUN_FINISHED = (45, 'Test case run finished on all slaves')

S_DEF_OK = (1, 'Test suite definition complete')

S_IDLE = (10, 'Waiting for cluster to activate')

S_INIT_ERROR = (-22, 'Test suite initialization error')

S_SLAVE_FINALIZED = (31, 'Slave finalized')

S_SLAVE_INITIALIZED = (21, 'Slave initialized')

S_SLAVE_TEST_FINALIZED = (47, 'Test case finalized on a slave')

S_SLAVE_TEST_INITIALIZED = (41, 'Test case initialized on a slave')

S_SLAVE_TEST_RUN_FINISHED = (44, 'Test case run finished on a slave')

S_WAIT_4_FINALIZE = (30, 'Finalizing test suite')

S_WAIT_4_INIT = (20, 'Initializing test suite')

S_WAIT_4_TEST_FINALIZE = (46, 'Finalizing test case')

S_WAIT_4_TEST_INIT = (40, 'Initializing test case')

S_WAIT_4_TEST_RUN = (43, 'Running test case')

checkIfDefComplete(*clusters*)
Makes sure all cluster definitions are complete.

@param clusters: all currently defined clusters.

getNextRunTime()
Get the next scheduled run time for this suite.

has_failure()

validateStatic()
Checks if definition (e.g given names) is statically correct.

exception XrdTest.TestUtils.**TestSuiteException**(*desc*, *typeFlag=1*)
Bases: exceptions.Exception

General exception raised by module.

ERR_CRITICAL = 2

ERR_UNKNOWN = 1

class XrdTest.TestUtils.**TestSuiteSession**(*suiteDef*)
Bases: XrdTest.Utils.Stateful

Represents run of Test Suite from the moment of its initialization. It stores all information required for test suite to be run as well as results of test stages. It has unique id (uid parameter) for recognition, because there will be for sure many test suites with the same name.

addCaseRun(*tc*)
Registers run of test case. Gives unique id (uid) for started test case, because one test case can be run many time within test suite session. @param tc: TestCase definition object

addStageResult(*state*, *result*, *uid=None*, *slave_name=None*)
Adds all information about stage that has finished to test suite session object. Stage are e.g.: initialize suite on some slave, run test case on some slave etc. @param state: state that happened @param result: result of test run (code, stdout, stderr, custom logs) @param uid: uid of test case or test suite init/finalize @param slave_name: where stage ended

getTestCaseStages(*test_case_uid*)
Retrieve test case stages for given test case unique id. @param test_case_uid:

sendEmailAlert(*failure*, *state*, *result=None*, *slave_name=None*, *test_case=None*, *timeout=False*)

XrdTest.TestUtils.**extractSuiteName**(*path*)
Return the suite name from the given path.

XrdTest.TestUtils.**loadTestCasesDefs**(*path*, *tests*)
Loads TestCase definitions from .py file. Search for getTestCases function in the file and expects list of testCases to be returned.

@param path: path for .py files, storing cluster definitions

XrdTest.TestUtils.**loadTestSuiteDef**(*path*)
Load a single test suite definition.

@param path: path to the suite definition to be loaded.

XrdTest.TestUtils.**loadTestSuiteDefs**(*path*)
Loads TestSuite and TestCase definitions from .py files stored in path directory.

@param path: path for .py files, storing cluster definitions

XrdTest.TestUtils.**readFile**(*path*)

XrdTest.TestUtils.**resolveScript**(*definition*, *root_path*)
Grabs a script from some arbitrary path and appends a set of util functions to it.

Utils Module

class XrdTest.Utils.**Command**(*cmd*, *cwd*)
Bases: object

Execute a subprocess command.

execute()

class XrdTest.Utils.**Logger**(*filename*)
Bases: object

Generic logging class

setup()

class XrdTest.Utils.**SafeCounter**
Bases: object

TODO:

get()

inc()

class XrdTest.Utils.**State**(*status_tuple*, *additDesc=''*)
Bases: object

Represents current state of some entity.

addDesc(*anyStr*)

isError()

class XrdTest.Utils.**Stateful**
Bases: object

Represents stateful entity and remember its previous states.

getState()

setState(*state*)

state

XrdTest.Utils.**redirectOutput**(*logFile*)
Redirect the stderr and stdout to a file

WebInterface Module

class XrdTest.WebInterface.**WebInterface**(*config*, *test_master_ref*)
All pages and files available via Web Interface, defined as methods of this class.

action(*type=None*, *testsuite=None*, *cluster=None*, *location=None*)

auth(*password=None*, *testsuite=None*, *cluster=None*, *type=None*)

clusters()

disp(*body*, *tvars*)
Utility method for displying a Cheetah template file with the supplied variables.

@param body: to be displayed as HTML page @param tvars: variables to be used in HTML page, Cheetah style

documentation()

downloadScript(**script_name*)
Enable slave to download some script as a regular file from master and run it.

@param script_name:

getSSSKeytable()

getSignedCertificate(***kwargs*)

getTrustedCACertificate(***kwargs*)

handleCherrypyError()

http_methods_allowed(*methods=['GET', 'POST']*)

hypervisors()

index()

indexRedirect()
Page that at once redirects user to index. Used to clear URL parameters.

showScript(**script_name*)
Enable slave to view some script as text from master and run it.

@param script_name:

testsuites(*ts_name=None*)

ts_vars(*ts_name*)

unsupported()

update(*path*)

vars()
Return the variables necessary for a webpage template.

exception XrdTest.WebInterface.**XrdWebInterfaceException**(*desc*)
Bases: exceptions.Exception

General Exception raised by WebInterface.

1.9.2 XrdTestMaster Module

class XrdTestMaster.**XrdTestMaster**(*configFile*, *backgroundMode*)
Bases: XrdTest.Daemon.Runnable

Main class of module, only one instance can exist in the system, it's runnable as a daemon.

activateCluster(*cluster*)
Start a cluster without attaching it to a particular test suite.

archiveSuiteSessions()

cancelTestSuite(*test_suite_name*, *timeout=False*)
Cancel a running test suite

checkIfSuitsDefsComplete()
Search for incompletness in test suits' definitions, that may be caused by e.g. lack of test case definition. @param dirEvent:

createCA()
Generate CA key/cert suitable for signing slave CSRs.

enqueueJob(*test_suite_name*)
Add job to list of jobs to run immediately after foregoing jobs are finished.

@param test_suite_name:

executeJob(*test_suite_name*)
Closure to pass the contexts of method self.fireEnqueueJobEvent: argument the test_suite_name.

@param test_suite_name: name of test suite @return: lambda method with no argument

finalizeTestCase(*test_suite_name*, *test_name*)
Sends runTest message to slaves.

@param test_suite_name: @param test_name: @return: True/False in case of Success/Failure in sending messages

finalizeTestSuite(*test_suite_name*)
Sends finalization message to slaves and destroys TestSuiteSession.

@param test_suite_name: @return: True/False in case of Success/Failure in sending messages

fireEnqueueJobEvent(*test_suite_name*)
Add the Run Job event to main events queue of controll thread. Method to be used by different thread.

@param test_suite_name: @return: None

fireReloadDefinitionsEvent(*type*, *dirEvent=None*)
Any time something is changed in the directory with config files, it puts proper event into main events queue. @param type: @param dirEvent:

getSuiteSlaves(*test_suite*, *slave_state=None*, *test_case=None*)
Gets reference to slaves' objects representing slaves currently connected. Optionally return only slaves associated with the given test_suite or test_case or being in given slave_state. All given parameters has to accord. @param test_suite: test suite definition @param slave_state: required slave state @param test_case: test case defintion

handleClientConnected(*client_type*, *client_addr*, *sock_obj*, *client_hostname*)
Do the logic of client incoming connection.

@param client_type: @param client_addr: @param client_hostname: @return: None

handleClientDisconnected(*client_type*, *client_addr*)
Do the logic of client disconnection.

@param client_type: @param client_addr: @return: None

handleClusterDefinitionChanged(*dirEvent*)
Handle event created any time definition of cluster changes. @param dirEvent:

handleSuiteDefinitionChanged(*dirEvent*)
Handle event created any time definition of test suite changes. @param dirEvent:

handleTagRequest(*slavename*)
TODO:

initializeTestCase(*test_suite_name*, *test_name*, *jobGroupId*)
Sends initTest message to slaves.

@param test_suite_name: @param test_name: @param jobGroupId: @return: True/False in case of Success/Failure in sending messages

initializeTestSuite(*test_suite_name*, *jobGroupId*)
Sends initialize message to slaves, creates TestSuite Session and stores it in python shelve.

@param test_suite_name: @param jobGroupId: @return: True/False in case of Success/Failure in sending messages

isJobValid(*job*)
Check if job is still executable e.g. if required definitions are complete.

@param job: @return: True/False

loadDefinitions()
Load all definitions of example clusters and test suites at once. If any definitions are invalid, raise exceptions.

loadSuiteSessions()

procEvents()
Main loop processing incoming MasterEvents from main events queue: self.recvQueue. MasterEvents with higher priority are handled first.

procSlaveMsg(*msg*)
Process incoming messages from a slave.

@param msg:

readConfig(*confFile*)
Reads configuration from given file or from default if None given. @param confFile: file with configuration

removeJob(*remove_job*)
Look through queue of jobs and remove one, which satisfy conditions defined by parameters of pattern job remove_job.

@param remove_job: pattern of a job to be removed

removeJobs(*groupId*, *jobType=6*, *testName=None*)
Remove multiple jobs from enqueued jobs list. Depending of what kind of job is removed, different parameters are used and different number of jobs is removed. @param groupId: used for all kind of deleted jobs @param jobType: determines type of job that begins the chain of jobs to be removed @param testName: used if removed jobs concerns particular test case @return: None

retrieveAllSuiteSessions()

retrieveSuiteSession(*suite_name*)
Retrieve test suite session from shelve self.suiteSessions @param suite_name:

run()
Main method of a programme. Initializes all serving threads and starts main loop receiving MasterEvents.

runTestCase(*test_suite_name*, *test_name*)
Sends runTest message to slaves.

@param test_suite_name: @param test_name: @return: True/False in case of Success/Failure in sending messages

runTestSuite(*test_suite_name*)
Run a particular test suite

selectHypervisor(*hypervisor=None*)

slaveState(*slave_name*)
Get state of a slave by its name, even if it's not connected. @param slave_name: equal to fully qualified hostname

startCluster(*clusterName*, *suiteName*, *jobGroupId*)
Sends message to hypervisor to start the cluster.

@param clusterName: @param suiteName: @param jobGroupId: @return: True/False in case of Success/Failure in sending messages

startNextJob()
Start next possible job enqueued in pendingJobs list or continue without doing anything. Check if first job on a list is not already started, then check if it is valid and if it is, start it and change it's status to started.

@param test_suite_name: @return: None

startTCPServer()
TODO:

startWebInterface()
TODO:

stopCluster(*clusterName*)
Sends message to hypervisor to stop the cluster.

@param clusterName: @return: True/False in case of Success/Failure in sending messages

storeSuiteSession(*test_suite_session*)
Save test suite session to python shelve self.suiteSessions @param test_suite_session:

watchDirectories()
TODO:

exception XrdTestMaster.**XrdTestMasterException**(*desc*)
Bases: exceptions.Exception

General Exception raised by XrdTestMaster.

XrdTestMaster.**main**()
Program begins here.

1.9.3 XrdTestHypervisor Module

class XrdTestHypervisor.**XrdTestHypervisor**(*configFile*, *backgroundMode*)
Bases: XrdTest.Daemon.Runnable

Test Hypervisor main executable class.

connectMaster(*masterName*, *masterPort*)
Try to establish the connection with the test master.

@param masterName: @param masterPort:

handleStartCluster(*msg*)
Handle start cluster message from a master - start a cluster.

handleStopCluster(*msg*)
Handle stop cluster message from a master - stop a running cluster.

readConfig(*confFile*)
Reads configuration from given file or from default if None given. @param confFile: file with configuration

recvLoop()
Main loop processing messages from master. It take out jobs from blocking queue of received messages, runs appropriate and return answer message.

run()
Main thread. Initialize TCP threads and run recvLoop().

tryConnect()
Attempt to connect to the master. Retry every 5 seconds, up to a maximum of 500 times.

updateState(*state*, *clusterName*)
Send a progress update message to the master.

exception XrdTestHypervisor.**XrdTestHypervisorException**(*desc*)
Bases: exceptions.Exception

General Exception raised by XrdTestHypervisor.

XrdTestHypervisor.**main**()
Program begins here.

1.9.4 XrdTestSlave Module

class XrdTestSlave.**XrdTestSlave**(*configFile*, *backgroundMode*)
Bases: XrdTest.Daemon.Runnable

Test Slave main executable class.

connectMaster(*masterName*, *masterPort*)
TODO:

executeSh(*cmd*)
@param cmd:

handleTestCaseFinalize(*msg*)
TODO:

handleTestCaseInitialize(*msg*)
TODO:

handleTestCaseRun(*msg*)
TODO:

handleTestSuiteFinalize(*msg*)
TODO:

handleTestSuiteInitialize(*msg*)
TODO:

parseTags(*command*)
TODO:

readConfig(*confFile*)
Reads configuration from given file or from default if None given. @param confFile: file with configuration

recvLoop()
TODO:

requestTags(*hostname*)
TODO:

run()
TODO:

exception XrdTestSlave.**XrdTestSlaveException**(*desc*)
Bases: exceptions.Exception

General Exception raised by XrdTestSlave.

XrdTestSlave.**main**()
Program begins here.

1.10 Appendix

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

PYTHON MODULE INDEX

C

X

INDEX

E

F

G

H

I

J

L

M

N

P

R

S