Remote Job Dispatcher

Accenture StormTest Development Center

Document ID: ST-13026

Revision Date: March 2016

Product Version: 3.3

Web: http://www.accenturestormtest.com

Contents

1 Preface

1.1 Accenture StormTest

Accenture StormTest Development Center is the leading automated test solution for digital TV services. It is designed to reduce the cost of getting high quality digital TV services to market faster.

StormTest Development Center greatly reduces the need for time-consuming, expensive and error-prone manual testing and replaces it with a more accurate and cost-effective alternative. It scales easily to large numbers and types of devices and integrates with existing infrastructure to give much greater efficiency in testing. It can be used to verify and validate services on a virtually every piece of consumer premises equipment (CPE), from set-top boxes to games consoles and from iPads to Smart TVs. It has been specifically designed to meet the needs of developers and testers of these CPE devices and the applications which run on them.

StormTest Development Center consists of:

- A choice of hardware units that can test 1, 4, or 16 devices. Each device under test can be controlled individually and independently and the audio/video from each device can be captured and analysed to determine the outcome of the test. The StormTest hardware supports capture of audio and video over HDMI interfaces and supports all HD resolutions up to 1080p. In addition there is a hardware upgrade option for the 16 device tester that will allow native capture of UHD content.
- Server software that controls all the hardware and devices in the rack as well as managing a central repository of test scripts and a central database of test results.
- A Client API that allows test scripts to interact with the server software
- A number of graphical tools that allow the user to directly control devices connected to StormTest Development Center, to create and schedule tests to run and to view the results of those test runs.

Test scripts can be run from any location – the tester needs only a network connection to the StormTest server. Video and audio output from the devices under test can be streamed over this network to any location, allowing remote monitoring and control of testing, either within a company LAN or across a WAN. Alternatively, scheduled tests can run directly on the server, negating the need for maintaining a continuous network connection to the StormTest server.

1.2 About This Document

This document describes the StormTest remote job dispatcher API.

This document is intended for the experienced script writer to allow them to programmatically interact with the StormTest scheduler feature.

1.3 Who Should Read This Document

This document is aimed at both

- Test developers and

- Test executors

That have a need to execute StormTest scripts using the StormTest scheduler, in a programmatic way

2 Getting Started

2.1 Installation

The standard StormTest client installation program installs all the components needed to use the remote job dispatcher API.

2.2 Architecture

The architecture of a typical StormTest system is shown in Figure 1. The core components are:

- StormTest Servers.

  One or more servers that interface with the StormTest hardware. These are responsible for controlling the devices under test (DUTs) and for monitoring the video, audio and serial data from the DUTs. The server could control 1, 4, or 16 DUTs

- Configuration Server.

  A single server which hosts the StormTest Database and file repository. It may be combined on the same physical machine as one of the StormTest servers. In a small system, it will almost certainly be combined with a StormTest server.

  All test scripts and test results are stored centrally on the configuration server.

- Scheduler

  This is a process which runs on the Configuration Server and dispatches tests to be executed by daemon processes.

- Client Daemon

  This is a process which runs on each StormTest server. It is a service which accepts jobs from the scheduler, launches the test and passes the result from that test back to the StormTest Database.

  Log files from test runs are stored locally on the specific machine that is hosting the client daemon service.

- StormTest Client API

  The standard StormTest client API, is the main programmatic interface to the StormTest platform.

- The user

  The most important part of the system – You!

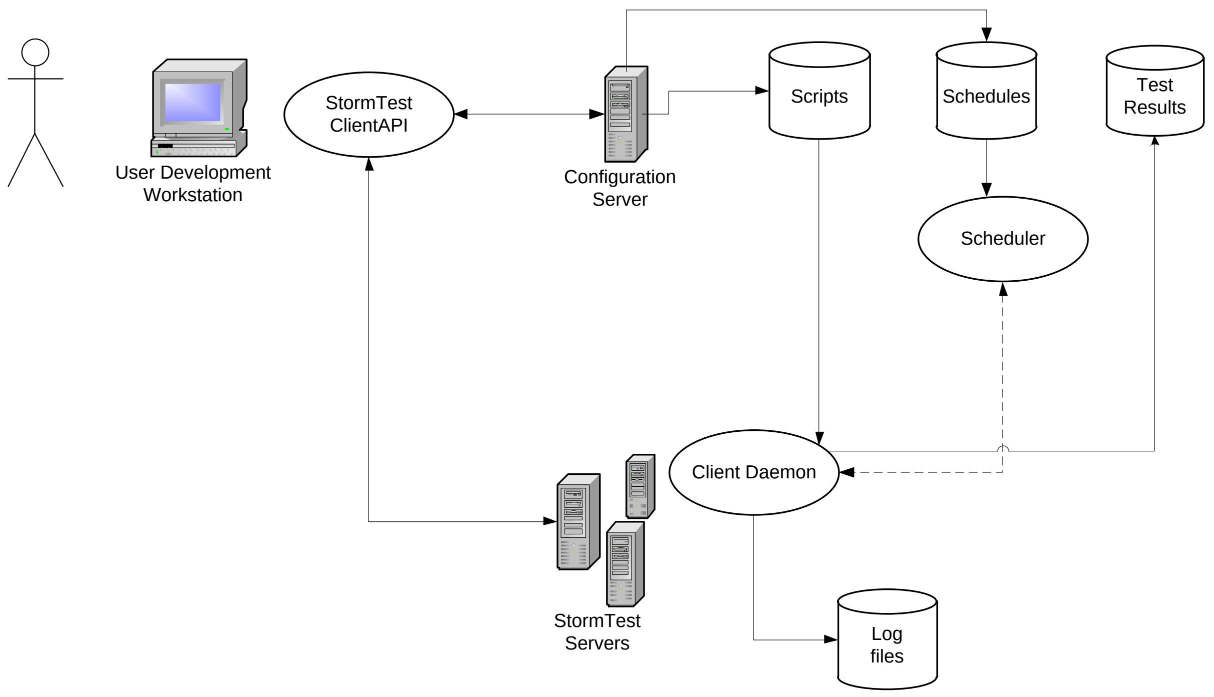

Figure 1 - System Architecture

3 Writing scripts to be executed by the scheduler

Writing a script to be run by the scheduler requires very little changes from writing a script to be invoked from the command line. In most cases, no changes are needed at all and the same script may be used.

3.1 ConnectToServer()

This function defines the server to be used. When a test is run under a daemon, the function must still be called but the value of the server name is ignored. The actual server will depend on the server that is assigned to the schedule by the scheduler. Thus, you should not assume in the script that because you choose a specific server in `ConnectToServer()` that your script is connected to that server.

If your script needs to know what server it is actually connected to at run time, it should use the `GetPhysicalAllocations()` API.

3.2 ReserveSlot()

This function selects which slot the test will run against. When running under a daemon, the function must still be called but the slot number passed in is ignored. The actual slot is allocated by the scheduler. Thus you should not assume in the script that because you chose a specific slot in ReserveSlot() that your script really is using that slot.

Again, if your script needs to know the actual slot being used, then use the `GetPhysicalAllocations()` API.

It should be noted that you may code your script to use a specific slot and pass that to other API functions that need a slot number. You may also rely upon the return values of API calls which return a slot number to return the same value as you specified in `ReserveSlot()`. This is particularly useful if your script uses 2 slots and you call `ReserveSlot()` twice with different slots – although you don't know which physical slots you are using, your script can be coded as if you do know – the API is fully internally consistent.

3.3 ReturnTestResult()

This function is used to return results into the database. It is important that the return result is one of the pre-defined values of the TM class – otherwise StormTest will not properly interpret the result.

3.4 Image Files

Often in a script, external image files are needed, for instance to compare with captures from the STB. When using these, path names relative to the location of the script (or relative to the location set by a call to `SetDirectories()`) must be used. Absolute path names should never be used as they cannot be guaranteed to be valid on the system that is actually hosting the daemon and executing the test.

Also, because the client daemon has its own storage, care should be taken that path names do not exceed the Windows limit of 260 characters – it is recommended that the internal path name for a script not exceed 200 so that some room is allowed for the directories used by the script runner.

3.5 SetDirectories()

The logDir parameter of the `SetDirectories()` call is ignored when a test is run under the client daemon. Using it will not cause an error but its value will be silently ignored – the log files will be placed where the daemon decides.

The actual path to all output files (log files, screen captures etc) will always be returned by the API calls that generate them.

4 Remote Job API

This section describes the remote job API and how to programmatically submit a script to run remotely. It is assumed that you are familiar with Python and the StormTest API.

To submit remote jobs, you must write a separate script from the normal StormTest script. This script will upload one or more tests to the server and configure the scheduler to run those tests.

All the examples below assume that your script has the following at the start:

```
from stormtest.ClientAPI import *

from datetime import *
```

For this guide, we are assuming that you have a script on your hard drive at

```
c:\samples\test1\script_one.py
```

and it uses an image at `c:\samples\test1\image_one.bmp`

4.1 Contacting the Configuration Server

To submit remote jobs, your script uses the configuration server. To establish contact, you must use the `SetMasterConfigurationServer()` call:

```
SetMasterConfigurationServer(“server”)
```

The “server” is the name or IP address of the configuration server as set up by your StormTest administrator.

Once you have specified the server, you can get the job configurator object, which gives access to the configuration server:

```
cfg = GetJobConfigurator()
```

To confirm the connection, you can get the version and server start time:

```
>>> print cfg.ServerVersion

0.0.3528.24534


>>> print cfg.ServerStartTime

2012-10-22 13:52:20Z
```

If the ServerVersion property has the value “Not Connected” then there is a problem connecting to the configuration server. The server start time is returned in UTC (hence the trailing Z) in the universal sortable format (YYYY-MM-DD HH:MM:SS) using the 24 hour clock.

4.2 Using the API from a non-Windows environment

In some cases, you may wish to use this API from a non-Windows environment, such as on a Linux machine. As the remote job dispatcher API is platform independent, this is perfectly possible, but as the ClientAPI module cannot be imported on a non-Windows python platform, you need to follow a different path to get access

4.2.1 Installation

The StormTest client installer won't work in this instance. However, you only need one file, so copy JobConfigurator.pyc from the C:\Python27\Lib\site-packages\stormtest directory on a Windows machine with StormTest installed onto your target machine in the appropriate site-packages directory. The target machine must be running python 2.7.x

4.2.2 Getting the Job Configurator Object

You're script needs access to the job configurator object in order to access the complete API. When running under Windows using the standard Client API, there is an API call to get this object. When running under Linux or Mac, you can get the same object by using code such as this:

```
from stormtest.JobConfigurator import CJobConfigurator
cfg = CJobConfigurator(server+":8001")
```

In the code above, replace 'server' with the hostname of your StormTest Configuration Server.

Once you have the *cfg* object, you can call any of the other APIs referred to in this document.

4.3 Uploading a script

There are 2 types of script, public scripts and private scripts. Public scripts can be accessed by any user but private scripts can only be used by the user who uploaded them.

To upload all the files we need for this test, the following call is used:

```
>>> cfg.uploadFiles (“c:\\samples”, “samples”, makePublic=True,
includeSubDirs=True)
```

This uploads everything in the c:\samples directory to a public directory on the configuration server which has the logical name “public/samples”. The absolute physical location is not exposed to the user and should not be relied upon by the test script.

The expected response would be:

```
[0, 'OK', ['\\samples', '\\samples\\test1', '\\samples\\test1\\image_one.bmp',
'\\samples\\test1\\script_one.py'], ['c:\\samples\\test1\\image_one.bmp',
'c:\\samples\\test1\\script_one.py'], []]
```

The 1st two values of the return are the same format for all of the scheduling API calls, a number (0 means success, negative numbers are warnings and positive numbers errors) and a textual representation of the number – in this example the call succeeded so the return is **0** and **OK**.

For the uploadFiles call, the next return value is the array of files successfully uploaded (including any directories created) using the server names. The next value is the list of files uploaded using the local file names – no directories are included here. The last value is an array of files that failed to upload – in this case it is empty. If a file is locked or you don't have privilege to read a file, it could be listed here.

The script can now be run remotely.

4.4 Submitting the job

To submit the job, you need to decide on which slot(s) and / or STB(s) it will run on. The function to submit the job takes 4 mandatory parameters and one optional parameter. A full example is shown below:

```
>>> slotsToUse = [("rack1", 4), ("rack1", 5), ("rack1", 6), ("rack1", 7)]

>>> testsToRun = [(""samples/test1/script_one.py", True, 4, 1, "")]

>>> cfg.submitBatchJob("My 1st job", slotsToUse, None, testsToRun)

[0, 'OK', 38407, []]
```

The above code sets up an array of 4 slots to use on a StormTest server with the address "rack1". The above uses 4 slots, numbers 4,5,6 & 7.

The second line of code sets up the job to run. Any number of test scripts can be submitted as one job. They will be allocated to the slots specified as needed. The testsToRun is an array of tuples. Each tuple has 5 mandatory elements, these are, in order:

1. Test script file name
2. Boolean indicating a public or private test
3. The number of slots to use for the test
4. A value indicating whether to use multi-process or single process mode (see section 4.6)
5. Arguments needed by the script

The final line submits the job. The return is an array of 4 elements. The 3rd element is a handle to the job – you will need it to retrieve the status of the job or to kill it before it has completed should you decide that is necessary. The final element of the return is a list of warnings that occurred during submission. They are returned as an array of tuples, containing a status code, message and the parameter that caused the warning. An example would be submitting a job on a slot where another user had also submitted a job. Both jobs will run but the later submitter may have to wait.

The optional parameter is a Boolean failOnWarning which defaults to false. If set to true, then any warning will prevent the job being started and the handle is never allocated.

4.5 Allocating DUTs

Instead of allocating a slot to a remote job, it is possible to allocate a specific device under test (DUT) to the job – this way, you can guarantee a script runs on a specific device, even if the device is moved to another slot. This is achieved by supplying an array of DUTs instead of slots to the submit batch job:

```
>>> dutsToUse = [11682, 3820, 4589, 983]

>>> testsToRun = [("“"samples/test1/script_one.py", True, 4, 1, “”)]

>>> cfg.submitBatchJob(“My 1st job”, None, dutsToUse, testsToRun)

[0, 'OK', 38407, []]
```

This uses DUT ids numbered 11682, 3820, 4589 and 983.

So where do the magic numbers come from? You can search for a DUT (or list of DUTs) by using the function:

```
>>> cfg.findStb([“Name==”TP*”, “Manufacturer==*”])

[0, 'OK', [['TP200686A-L-80', 'Uses ftp bootloader', '', 'Model Name', 'Model
Description', 'Manufacturer', 'Manf’s Model Name', '07', 'HD+', 'HD PVR Box',
'Broadcaster', 'IR Type, 'IR description', 'SmartCard s/n', 'NULL', 'NULL', 3822,
1247498326, 16, 'rack1']]]
```

This results in a list of matching DUTs. You can then decide which one(s) you want. In the above example just one device is returned. The information returned is listed below. Note that this information is the data configured using the StormTest Admin Console. If the fields are left blank in the Admin Console, then they will also be blank here.

- Name of device
- Description of device
- Serial Number of device
- Model Name of device
- Model Description of device
- Manufacturer of device
- Manufacturers model Name for device
- Hardware version of device
- Generic Type of device
- Generic Description of device
- The broadcaster to which the device is tied
- The type of IR remote control for the device
- The description of the remote control for the device

- The Smartcard serial number
- The 1st attribute name of the Smartcard
- The 1st attribute value of the Smartcard
- The ID of the device
- The creation date and time of the IR definition (in seconds since 1/1/1970 00:00:00)
- The slot number where the device is currently located
- The server where the device is currently located.

So, a simple script to add all the DUTs made by a specific manufacturer would be:

```
dutsToUse = []

manfName = “My Favourite Manufacturer”

devicesFound = cfg.findStb([“Manufacturer==” + manfName])[2]

for device in devicesFound:

       stbsToUse.append(device[16]) # device id is the 16th element
```

`findStb()` allows searching on many fields with a variety of options (the main one being the wildcard (*) operator). The full list of fields that can be searched are:

"Name", "Description", "SerialNumber", "ModelName", "ModelDescription", "Manufacturer", "ManufactureModel", "HardwareVersion", "TypeName", "TypeDescription", "Broadcaster","IRName", "IRDescription","SmartcardNumber", "SmartcardParamName", "SmartcardParamValue", "ModelId"

4.6 Multi Process vs Single Process

When you submit a test to run on more than one slot, two options exist:

- The test could execute using 1 process and be allocated all the slots on the expectation that it would call `ReserveSlot()` multiple times
- The test could use multiple processes, each of which is allocated one slot and calls `ReserveSlot()` once.

Either method can be used to write a test to run on more than one slot. However, the second method makes the tests much easier to write and means that there is a unique result for each test run on each slot. As a result ‘multi-process’ mode is the preferred and recommended method to use when scheduling tests.

Single process mode should only be used in very specific circumstances and would be considered appropriate for ‘expert’ StormTest script developers. As such, we will not consider it any further in this guide.

The 4th member of the tuple when setting up the tests to use should be 0 or non-zero to use the single process / multiple process mode. Please ensure that this is not set to 0 or your tests may not run on all the slots you intend them to run on.

4.6.1 Multi-process Coverage Modes

When choosing multi-process mode, there are actually two choices which determine the order in which the tests are scheduled.

- Option 1: DUT Coverage. In this mode, each test will be run on each DUT in order. So, when the schedule starts, test 1 will start on DUTs 1,2,3,4 etc. When test 1 finishes, test 2 will start and so forth. This may mean it takes a long time before the last test in the list is reached.
- Option 2: Test Coverage. In this mode as many tests are run as possible without duplicates, thus ensuring that all tests are executed as soon as possible. So when a schedule starts, test 1 runs on DUT 1, test 2 on DUT 2, test 3 on DUT 3 etc. When test 1 is complete, DUT 1 will take the next unexecuted test – it will not take test 2 until all tests have been executed at least once.

To select DUT Coverage, set mode to **1**. To select Test Coverage, set mode to **2**.

4.7 Killing a job

Sometimes it may be necessary to kill a remote job before it has finished. To do this:

```
>>> cfg.killBatchJob(38407)

[0, ‘OK’,38407]
```

Where the parameter to `killBatchJob()` is the handle received from `submitBatchJob()`.

4.8 Private Scripts

StormTest can upload private scripts, simply by changing one parameter of the `uploadFiles()` function call: makePublic=False. In fact, this is the default setting if the parameter is omitted. Each user is allocated a separate private files area. It is not possible for one user to use another user’s private files. It is also not possible for a public script to use a user’s private files. During initial script development where there are many changes to a script, there are considerable advantages to using private scripts instead of public scripts.

When submitting a test, a flag indicates whether the test is public or private. If a file is uploaded to a private area, then the flag in the `submitBatchJob()` call for that test must be set to False (to indicate non public) otherwise the system won’t be able to find the file – this is considered a warning so the `submitBatchJob()` will continue but not run the test.

4.9 Using public files in private scripts

At times it is very useful to use public files within a private script, for instance, if the public area has a library of image files then it would be useful to be able to use those images from within a private script. This is possible by using the variable STORMTEST_PUBLIC. When a script is running as a batch job, this variable is the absolute path to the public files area. So, if a file is uploaded to the public area "images\image1.bmp" then it can be used within a script (private or public) as:

```
os.path.join(STORMTEST_PUBLIC, "images\\image1.bmp")
```

assuming that there is a prior import os.path statement in the script. If the script is running interactively from the command line then STORMTEST_PUBLIC will be None – your script will need to handle this case, maybe by checking at the start of the script and setting it to some specific value. Continuing the example from above, if the local file was in c:\work\images\image1.bmp" then at the start of the script the following statement could be used:

```
if STORMTEST_PUBLIC == None:
        STORMTEST_PUBLIC = "c:\\work"
```

Now later you can always find the image, whether run remotely or interactively by using:

```
os.path.join(STORMTEST_PUBLIC, "images\\image1.bmp")
```

4.10 Converting Private Scripts to Public

Once a private script is working, it is often useful to convert it to a public script so that other users may use the script. This is achieved by a single function call:

```
>>> cfg.convertFilesToPublic("/path/to/private/*.py", "/public/path/",
includeSubDirs = True)
```

It is possible to convert a single file or a whole tree. Note that this does not delete the private files – they remain available for use. Updates can then be made to the private files and, if needed, converted again to be public – the latest version of the private file is then made public.

4.11 User Names

The private scripts are stored "per user" – the user name is, by default, the user name used to log on to Windows. This can be changed via the `setUser()` call:

```
>>> cfg.setUser("somename")
```

The current release places no restrictions on user name nor does it enforce any privileges (later versions may do this) so it is possible for any user to impersonate any other user. The call to `setUser` can be made at any time and multiple times.

5 Viewing Results

To view the results, either a Python script can be used or the results can be viewed via the StormTest Developer Suite. Also, the results are available via the StormTest a reporting tool configured to use the database is necessary. A reporting tool can generate trends and statistics across a lot of tests.

5.1 Getting Status of the job

From a Python script a simple command:

```
>>> cfg.getBatchStatus(38407)
```

Status Start End

Detailed test status

```
[0, 'OK',
   [2, 1257767625, 1257767630, 0,
      [
        [38413, 0, 111, 'Test has run to completion', 1257767626, 1257767627,
        'public/samples/test1/script_one.py',
        'ftp://rack1:21/public/samples/test1/script_one_20091109_115346',
        [['192.168.7.60', 1, 149]]],

        [38413, 0, 111, 'Test has run to completion', 1257767627, 1257767628,
        'public/samples /test1/script_one.py', 'ftp://rack1:21/public/samples
        /test1/script_one_20091109_115347', [['192.168.7.60', 2, 150]]],

        [38413, 0, 1, 'Incomplete test', 1257767629, 1257767629, 'public/samples
        /test1/script_one.py', 'ftp://rack1:21/public/samples
        /test1/script_one_20091109_115349', [['NULL', 'NULL', 'NULL']]],

        [38413, 0, 1, 'Incomplete test', 1257767630, 1257767630,
        'public/samples/test1/script_one.py', 'ftp://rack1:21/public/samples
        /test1/script_one_20091109_115350', [['NULL', 'NULL', 'NULL']]]
      ]
   ]
]
```

This is the usual 3 element array with the 3rd element being the detailed status. This is structured as an array of 5 elements:

1. Status of job, One of

    0 = Idle (short lived after submitBatchJob is called)
    1 = Running
    2 = Completed
    3 = Stopped by user via killBatchJob()
    4 = Stalled – the StormTest client daemon has stopped temporarily

5 = Could not start, no daemon on the specified server
6 = Could not start, no DUTs in specified slots (or DUT IDs not in system)
7 = Could not start, multiple servers needed to use the slots specified.

2. Time job started, in seconds since 1/1/1970 (Unix epoch)
3. Time job ended, in seconds since 1/1/1970 – 0 is still running
4. Number of tests still to run
5. Array of detailed test statuses

A note on status code 7: when submitting **single process mode** job, all the specified slots and DUTs must be on the same server. A single process mode test cannot reserve slots that span multiple servers.

Each element of the test status array represents the result of 1 test. It is a 9 element array with elements:

1. Test Identifier – in multi-process mode, each process of the test uses the same test identifier so you can match them up. It has no other significance.
2. Test Status, one of:

   0 = complete
   1 = running now
   2 = waiting to run

3. Return status from the test – see StormTest API for numeric code. 0 if test not complete
4. Comment – human readable interpretation of status
5. Test Start time in seconds since 1/1/1970. Will be 0 if not yet started
6. Test End time in seconds since 1/1/1970. Will be 0 if not yet finished.
7. Path to script – identifies the test that ran
8. URL to logs – URL where the output of the test (logs, images, videos etc) can be found
9. Array of slots where the test ran (will be empty if test has not started).

The array of slots will have one element for those tests which run on one slot, otherwise it will have multiple elements. Each element is a tuple of 3 sub elements:

1. StormTest server address used
2. Slot number
3. DUT identifier in the slot.

In the example shown, the first two tests finished correctly and passed (status code 111). The last two tests finished with a status code of 1 and a comment indicating “Incomplete Test”. The slot list was also [NULL, NULL, NULL] – this indicates that the test did not actually execute for some reason.

6 System Considerations

6.1 File Updates

Previous sections have shown how files are uploaded and then used to run tests. However, an important issue was overlooked: how do you know which files are used for tests and what happens if you upload a file that already exists? What happens if you upload a file that is being used by another user's test?

If a file is uploaded that already exists, then the existing file is replaced. No error or warning is given.

However, as a test may be using that file and StormTest has no way of knowing if this is the case or not, it delays the update in an intelligent manner to ensure that already running tests are not disrupted and later submitted tests will use the new file. The exact algorithm depends on whether a new file is public or private

6.1.1 Updating a public file

Once any user uploads a new public file, all pending tests are put into a "waiting state". All currently running tests (both public and private) are allowed to complete before the working copy is updated. Once the update is complete, pending tests can execute in the working copy again. Note that each daemon can have more than one working copy (default is 4) which means that this delay is unlikely to affect users who are scheduling tests in a real world scenario, as there will almost always be a fully updated working copy available to take the job.

6.1.2 Updating a private file

Since public tests do not use private files and each user has their own private area, an upload of a private file puts all pending tests for that user only into a "waiting state", all currently running tests for that user are allowed to complete. Only then, is the change made permanent, and the pending tests resume.

This means that an upload of a private file only disrupts tests by the user doing the upload only. The assumption is that if a change is made, then all later tests would want to use the new file(s). For this reason, during initial development, private files and tests are less disruptive to other users and are recommended.

6.2 CPU loading

An attempt is made to ensure that not too many tests are launched at the same time. Before starting a new batch job, a measurement of the current CPU usage is made and if it is above a threshold (80% by default) then the test will not be dispatched. Instead, it will be attempted again in 30 seconds.

If there is more than one client daemon in your StormTest facility, then a daemon on another server can be used, assuming it is not overloaded. This decision is transparent to the user and is managed by the Scheduler.

7 Advanced API usage

The API as described in this document is used to submit jobs to StormTest that are executed as soon as possible.

The job configurator object exposes a number of other APIs that allow the user access to other types of schedules and that allows existing schedules to be modified programmatically. This is considered a more advanced usage of the API and could result in broken schedules. As a result, these APIs must be used with care.

7.1 Available APIs

The following is a list of available APIs. In most cases, the purpose of the API is obvious. Where it is not, a brief description is included.

For all APIs, the return value will be a tuple of the form:

[X, ‘Y’, Z]

Where:

- X is an integer error code, with 0 indicating success.
- ‘Y’ is a status string, such as ‘OK’ indicating in a readable fashion whether the API encountered an error or not.
- Z is some return information specific to the API call. In some cases this element will not be present.

7.1.1 createSchedule

```
def createSchedule(name, scheduleType, startDate, endDate,
                   startTime, endTime, repeatCount=0, repeatRate=1,
                   repeatDays = None):
```

Create a new schedule. Once the schedule is created, the user must then add either slots or DUTs to the schedule and also add jobs (test scripts) to the schedule. The parameters used by this API are:

scheduleType: Loop | OneOff | Daily | Weekly | Weekdays | Weekends
startDate / **enddate** is a python datetime.date object, for example `datetime.date( 2013, 9, 17 )`
startTime / **endTime** is a python datetime.time object (may have timezone. If not, StormTest server time is used). For example datetime.time(10, 0) # 10am local time
repeatCount: How many times to iterate the schedule
repeatRate: the frequency of iteration – only really used for ‘Daily’ schedules.
repeatDays: needed only for Weekly and must be string list of English Language day abbreviations such as ["Mon", "Wed", "Fri"]

For a ‘OneOff’ schedule, startDate + startTime specifies the start of the schedule and endDate + endTime the end of the schedule.

If repeatCount is specified, then the enddate must be set to None.

Similarly if enddate is set, not not set repeatCount. If enddate is set to None and repeatCount is omitted, then the schedule will have no end set.

The return value is a tuple: errorcode, status string and schedule ID (as a string). For example:

```
print cfg.createSchedule('schedule name','Daily',
                         datetime.date(2013,9,17), None,
                         datetime.time(10,0), datetime.time(20,0),
                         100,
                         2)
[0, 'OK', '1103709']
```

7.1.2 renameSchedule

```
    def renameSchedule(scheduleId, newName):
```

For example:

```
print cfg.renameSchedule(1103713,'new name')
[0, 'OK']
```

7.1.3 modifySchedule

```
    def modifySchedule(scheduleId, scheduleType, startDate, endDate,
                       startTime, endTime, repeatCount=0, repeatRate=1,
                       repeatDays = None):
```

Similar to createSchedule() but with no name. Also, the ID of the schedule must be known. For example:

```
print cfg.modifySchedule(1103713, 'Daily',
                         datetime.date(2013,9,17), None,
                         datetime.time(10,0), datetime.time(20,0))
[0, 'OK', '1103713']
```

7.1.4 deleteSchedule

```
    def deleteSchedule(scheduleId):
```

Example:

```
print cfg.deleteSchedule(1103713)
[0, 'OK']
```

7.1.5 getScheduleIdFromName

```
    def getScheduleIdFromName(name, userName=""):
```

Returns a schedule ID. If username is not specified, then it will default to the logged in user.

```
print cfg.getScheduleIdFromName('a weekday schedule')
[0, 'OK', 1103266]
```

7.1.6 addSlotToSchedule

```
    def addSlotToSchedule(scheduleId, serverName, slotId):
```

Adds a server/slot pair to the schedule.

```
print cfg.addSlotToSchedule(1103266,'stormtest16',3)
[0, 'OK']
```

7.1.7 removeSlotFromSchedule

```
    def removeSlotFromSchedule(scheduleId, serverName, slotId):
```

Removes a server/slot pair from the schedule.

```
print cfg.removeSlotFromSchedule(1103266,'stormtest16',3)
[0, 'OK']
```

7.1.8 addStbToSchedule

```
    def addStbToSchedule(scheduleId, dutId):
```

Add a DUT to the schedule. You can get the 'dutId' by using the findStb() API call described earlier in this document.

```
print cfg.addStbToSchedule(1103266,182)
[0, 'OK']
```

7.1.9 removeStbFromSchedule

```
    def removeStbFromSchedule(scheduleId, dutId):
```

For example:

```
print cfg.removeStbFromSchedule(1103266,148)
[0, 'OK']
```

7.1.10 deletePublicFile

```
    def deletePublicFiles(file):
```

Will delete a specific file or directory – no wildcards allowed. Note that if the argument is a directory, then that directory and all contained files/directories will also be removed.

```
print cfg.deletePublicFiles(a_dir\\a_test.py')
[0, 'OK', 'a_dir\\a_test.py']
```

7.1.11 deletePrivateFile

```
    def deletePrivateFiles(file):
```

Will delete a specific file or directory – no wildcards allowed. Note that if the argument is a directory, then that directory and all contained files/directories will also be removed

```
print cfg.deletePrivateFiles('test3.py')
[0, 'OK', 'test3.py']
```

7.1.12 addTestToSchedule

```
    def addTestToSchedule(scheduleId, testPath, publicJob, slotsNeeded,
                          priority=1, schedulingMode=0, overrideArgs=""):
```

Adds a test (also known as a 'job') to an existing schedule.
The parameters are the same as those used in the call to submitBatchJob(). Please remember to set slotsNeeded to the number of slots that you want that test to run on (normally this is the number of slots allocated to the schedule) and set schedulingMode to 1 – multi-processes mode.

Also, the default priority will be 1 (the highest priority). If you wish the job to have a lower priority, then you must specify that.

```
print cfg.addTestToSchedule(1103266, 'pass1.py', publicJob=False,
                            slotsNeeded=3, schedulingMode=1)
[0, 'OK', '1103735']
```

The returned value includes the job ID (the third element), which can be used in other API calls related to jobs.

7.1.13 updateJob

```
    def updateJob(jobId, publicJob, slotsNeeded, priority,
                  expectedDuration, schedulingMode, overrideArgs):
```

Used to modify the settings for a job. Note that all arguments are needed, although the value passed for expectedDuration is ignored.

```
print cfg.updateJob(1103735, publicJob=False, slotsNeeded=4, priority=5,
                    expectedDuration=0, schedulingMode=1,
                    overrideArgs='-s "an argument"')
[0, 'OK', '1103735']
```

7.1.14 DeleteJob

```
    def deleteJob(jobId):
```

Delete a job.

```
print cfg.deleteJob(1103735)
[0, 'OK', '1 records deleted']
```

7.1.15 showSchedules

```
    def showSchedules(self, userName=""):
```

Show all the schedules belonging to a user. By default, show the current logged in user's schedules.

The return value is the usual tuple, but the third element contains all the schedules, for example:

```
schedule_result = cfg.showSchedules()
schedules = schedule_result[2]
for schedule in schedules:
    print schedule

[1053445, 'a loop schedule', datetime.datetime(2013, 7, 12, 9, 10, 39), 'Loop',
datetime.date(2013, 7, 15), datetime.date(2013, 7, 19), datetime.time(0, 0),
datetime.time(0, 0), 1000, 'NULL', [], datetime.datetime(2013, 7, 15, 8, 0),
datetime.datetime(2013, 7, 19, 16, 0)]
[1103725, 'a daily schedule', datetime.datetime(2013, 9, 17, 12, 10, 19), 'Daily',
datetime.date(2013, 9, 17), datetime.date(2013, 12, 25), datetime.time(10, 0),
datetime.time(20, 0), 100, 1, [], datetime.datetime(2013, 9, 17, 9, 0),
datetime.datetime(2013, 12, 25, 10, 0)]
[1103262, 'a weekly schedule', datetime.datetime(2013, 9, 12, 14, 24, 8),
'Weekly', datetime.date(2013, 9, 12), datetime.date(2038, 1, 19),
datetime.time(15, 24), datetime.time(16, 24), 0, 3, ['Wed', 'Fri'],
datetime.datetime(2013, 9, 12, 14, 24), datetime.datetime(2038, 1, 19, 3, 14, 7)]
```

Information on the schedule includes:

- schedule ID
- schedule name
- Date/time created
- schedule type
- Start date
- End date (if no end date was set, then this will be 19/1/2038)
- Start time
- End time
- Number of times to repeat the schedule before stopping
- The frequency of iteration (i.e. every 2 days)
- List of days (weekly schedule only)
- Combined start date/time
- Combined end date/time

7.1.16 showSlotsOnSchedule

```
    def showSlotsOnSchedule(scheduleId):
```

Shows all slots allocated to a schedule. The third element of the return value is a list of server, slot pairs allocated to the schedule.

```
print cfg.showSlotsOnSchedule(1103742)
[0, 'OK', [['stormtest16', 2], ['stormtest16', 3]]]
```

7.1.17 showStbsOnSchedule

```
    def showStbsOnSchedule(scheduleId):
```

Shows all DUTs allocated to a schedule. The third element of the return value is a list of DUTs, showing the DUT ID, the DUT name, DUT description and the DUT model name.

```
print cfg.showStbsOnSchedule(1103742)
[0, 'OK', [[148, 'Grundig-1', '', 'gufsat20sd'], [149, 'Grundig-3', ' ',
'gufsat20sd']]]
```

7.1.18 showJobsOnSchedule

```
    def showJobsOnSchedule(scheduleId):
```

List all the jobs allocated to a schedule. The third element in the return value is the list of jobs. For example:

```
jobs_result = cfg.showJobsOnSchedule(1103742)
jobs = jobs_result[2]
for job in jobs:
    print job
[1103808, 1, 28744, datetime.datetime(2013, 9, 18, 10, 39, 57), '', 2862,
'public/tests/test1.py', 'ran', 5, 0, 1, 2, 1, 1379501100, 1379501192]
[1103809, 0, 28744, datetime.datetime(2013, 9, 18, 10, 39, 57), '', 2862,
'user1/Basic_Test.sttc', 'running', 5, 0, 0, 3, 1, 1379501101, 1379501184]
```

For each job, the data returned in the list is:

- Job id
- Loops completed
- User id of user that created job
- Date/time created
- Arguments
- Repository revision
- Path to test script
- Status (readable string)
- Priority
- Expected duration (this is unused)
- 1 = Public, 0 = private
- Number of slots needed for test
- Scheduling mode (1 = multi-process)
- Last start time in Unix time
- Last end time in Unix time

7.1.19 listPublicFiles

```
    def listPublicFiles(remotePath, includeDirs=False):
```

List all the files at the given path in the public area. To see directories as well, set includeDirs to True.

```
print cfg.listPublicFiles('\\tests')
[0, 'OK', [['test1.py', 1367572077, 11270, 'File'], ['test2.py', 1367572077,
11270, 'File']]]
```

For each file or directory, the following is returned:

- Name
- Last time the file was edited (in Unix time)
- Size in bytes
- Type (i.e. ‘File’ or ‘Dir’)

7.1.20 listPrivateFiles

```
    def listPrivateFiles(remotePath, includeDirs=False, userName=""):
```

List all the files at the given path in the user’s private area. The information returned on each element is the same as for listPublicFiles().

7.1.21 setUser

```
def setUser(userName=""):
```

Sets the username – allows the script to login as any user on the system. Currently, no access control is implemented. If no username is given, the logged in Windows username is used (this is the default case where this API has not been called).