UTAH Documentation

Release dev

Canonical Ltd

December 03, 2014

Contents

The Ubuntu Test Automation Harness project serves two purposes:

1. A test runner that can take a UTAH test definition, execute the tests, and provide test results. This is most commonly referred to as the *"utah client"*.
2. A mechanism to provision pristine systems which can then execute a UTAH test definition using the runner. There is provisioning support for:
    - Physical x86 servers and desktops using cobbler
    - Virtualized x86 servers and desktops using libvirt and KVM
    - Ubuntu Touch devices using phablet-tools

CHAPTER 1

Getting Started

Most user's of UTAH will be interested in creating a test suite. This requires learning about:

- How tests are defined
- How to run tests locally

Table of Contents

2.1 Test Definition Format

Any binary that can run on Ubuntu, can be considered a Test Case as far as UTAH is concerned. If the right control files are added and the test suite is structured in directories neatly, UTAH will run it.

2.1.1 Directory Structure

This is what a UTAH test suite looks like:

```
testsuite1/
  tslist.run (or tslist.auto)
  ts_control
  testcase1/
    tc_control
    Makefile
    test.c
  testcase2/
    tc_control
    test.py
```

The utah-client package includes a utility called phoenix that helps create a skeleton layout.

2.1.2 master.run

A runlist is a collection of test cases that we want to run in one go. The syntax for runlists is:

```
- testsuites:
  - name: testsuite1
    fetch_method: git
    fetch_location: repo

  - name: testsuite2
    fetch_method: bzr
    fetch_location: lp:utah/dev/
    include_tests:  # optionally include specific tests
      - t1
      - t2
      - t3...
    exclude_tests: # optionally exclude specific tests
```

```
    - st4
    - st5
```

The only required fields are `name`, `fetch_method` and `fetch_location`. `name` must correspond to the name of the top-level testsuite directory. `fetch_method` should be one of `bzr`, `bzr-export`, or `git`. `fetch_location` should be a valid location for the supplied `fetch_method`. If `bzr` is selected as the fetch method then `fetch_location` should point to a repository that is a valid testsuite, i.e. has a tslist.run, ts_control (if needed), and test case directories. While `bzr` does a `bzr branch` in the implementation, `bzr-export` does a `bzr export` and accepts a bzr location that can point to a sub-directory within a repository that is a valid testsuite. If `dev` is selected as the fetch method then `fetch_location` should point to a valid testsuite directory. The utah client will run `cp -r <fetch_location> <testsuite_name>`. This method is provided to allow testsuite/testcase authors to run the client on a development tree without needing to push changes each time to a repository.

One caveat to note is that a `fetch_method` of `bzr` will get the revision information from the local copy of the branched repository but `bzr-export` will have a small race condition between the `get` and `revision` calls since the actual bzr repository must be queried for the revision.

`include_tests` will limit the testcases that are run to only those listed by this option. `exclude_tests` will run all the tests in the testsuite's tslist.run file except those listed by this option.

Test suites can be divided in categories (for our internal use, or for test cases submitted to our test case base) or they can be in a repository anywhere.

repeating a runlist:

Sometimes a runlist may need to be executed more than once. For example, in the case you want to determine statistical accuracy of results. A runlist supports an optional field:

```
repeat_count: <integer>
```

This value defaults to 0 which means execute the runlist once. If non-zero the runlist will be repeated that many times. eg, a repeat_count of 2 means the runlist will be executed 3 times.

Each test suite has a default runlist and a control file that contain as follows:

2.1.3 tslist.run

```
- test: t1                              # directory
  overrides:
  - run_as: nobody                      # user that runs the test

- test: t2                              # directory/binary
  overrides:                            # array of control properties to override
  - timeout: 200                        # timeout in seconds
```

`test` is the only required field in this file and must match a directory name in the testsuite tree. `run_as` utilizes sudo to run the testcase as the given user. NOTE: this requires that the user running the utah client either has cached credentials or is allowed to run commands without a password. The `overrides` array accepts any option from the tc_control file. Overrides should be used sparingly since it makes more sense to adjust the options in the tc_control file rather than here.

2.1.4 tslist.auto

tslist.auto is an alternative to the static nature of tslist.run. When trying to add an existing test suite to UTAH, this option may work better.:

```
# the contents of this file help dynamically construct the names of each
# existing test case and how to run it.
# utah execute's the discovery command, and then pass each line of the output
# as a cli parameter(s) to the test_cmd
-
  # this will result in testcases name "1", "2" and "3". The output will be
  # will just be "1", "2", and "3"
  discovery_cmd: seq 3
  # the '{}' in test-cmd will be replaced with the output of the discovery
  # command.
  test_cmd: echo {}

-
  # a more complex example that breaks up unity test for autopilot
  discovery_cmd: "autopilot list unity | grep unity.tests | sed -e 's/^[[:space:]]*//' | cut -d. -f1,
  test_cmd: autopilot run -v {}
```

NOTE: The items returned by the discovery_cmd become the name of each testcase. So its wise to choose something sensible/consistent.

2.1.5 ts_control

```
build_cmd:      make
timeout:        300
ts_setup:       tsetup/setup.sh      # tsetup/setup.sh is the set up for
                                     # the whole test suite
ts_cleanup:     tcleanup/cleanup.sh  # clean up test suite
```

There are no required fields in this file.

To avoid excessive typing and repetition in runlists and control files the following rules apply:

- `tslist.run`'s **test** is relative to the test suite. So if the suite is called *sample_tests* and the entry in `tslist.run` for test is *test_one* then the path to the test folder will be *sample_tests/test_one*.
- `tslist.run`'s **overrides** are the same options available in the `tc_control` file and take precedence over those found in `tc_control`.
- `ts_control` is optional.
- if `build_cmd`, or `ts_setup` fails no tests in the suite will be run.
- if `build_cmd`, or `tc_setup` fails the test will not be run.
- `ts_cleanup` will be run whether or not `ts_setup`, `build_cmd`, or any tests fail.
- `tc_cleanup` will be run whether or not `tc_setup`, `build_cmd`, or the test fails.

Each test case, with all the code required for it to compile or run, lives in a directory, and contains also a control file that **includes its documentation** amongst other things:

2.1.6 tc_control

```
build_cmd:      make    # or scons or build.sh
command: python test1.py a1 a2 a3  # command to run the test from within t1
description:
 dependencies: coreutils
 actions: |
```

```
   1. Action 1
   2. Action 2
 expected_results: |
   1. Expected result 1
   2. Expected result 2
type:           userland/kernel    # currently unimplemented
timeout:        100
tc_setup:       t1/t1 setup    # in case the test case has a setup
tc_cleanup:     t1/t1 cleanup  # in case the test has a cleanup
```

In this file `command`, `description`, `dependencies`, `actions`, `expected_results`, and `timeout` are required. `command` is the command to actually run the test. `description` is a textual description of the test. `dependencies` is a list of items the test depends on. `action` is a list of actions the test executes and `expected_results` is a list of what those actions should result in. `timeout` is the ammount of time the test should complete within and is used to avoid tests that might loop indefinitely.

`type` currently defaults to `userland` and may in the future have other options as the need arises.

`build_cmd` can be used to build any binary test cases. `tc_setup` can be used to setup any needed data or files for the test and `tc_cleanup` can be used to remove the data or files added by `tc_setup`. `build_cmd`, `tc_setup`, and `tc_cleanup` are all simple shell commands similar to `command`.

For each test case that is a binary or a script, there will be an option to define a setup and a cleanup function if they wish so, and it is the test code developer's responsibility to add the right parameters to their code so that the harness can run their setup and clean up functions.

In terms of settings precedence: The test case control file (`tc_control`) provides the default values for the options available for the testcase. The **overrides** in `tslist.run` take precedence over those options in the `tc_control` file.

Tests That Need Reboots

In a situation where a reboot is required. This is handled in UTAH by marking a testcase (in the `tc_control` file) with:

```
reboot: <always, pass, never>
```

This will make the **utah client** reboot the system under test at the **end** of the testcase after any `tc_cleanup` command has finished. Any thing that needs to be tested after the reboot should be in the testcase immediately following the testcase marked for reboot.

The possible choices are `always`, `pass`, and `never`. `never` is the default and simply means the system under test will not reboot. `pass` means the reboot will only happen if the testcase passes. `always`, means just that, the reboot will happen even if the test fails.

Some things to keep in mind. The **utah-client** resumes by replacing `/etc/rc.local` with a file that will resume the current run and will resume with the testcase following the one that triggered the reboot. Also note that in order to have both pre- and post-reboot logging the `-o` flag should be used so that post-reboot output will be appended to the pre-reboot output.

2.2 Creating Testsuite and Testcase Skeletons

The utah-client package includes a utility called phoenix that helps create a skeleton layout. It can be used like this:

```
phoenix example_testsuite example1 example2
```

This creates a testsuite directory `example_testsuite` in the current directory with two testcase directories `example1` and `example2` inside. In addition a tslist.run, ts_control, and tc_control files for each testcase. A master.run file is created as well to aid in development. NOTE: the master.run file will generally be written by a person wishing to run a set of testsuites and/or testcases and will not be a part of a testsuite.

To add a testcase to an existing testsuite:

```
phoenix example_testsuite example3
```

This will add an `example3` testcase directory to the example_testsuite testsuite as well as adding an entry to the testsuite's tslist.run. From within a testsuite directory this can be shortened to:

```
phoenix . example3
```

2.3 Running Tests

2.3.1 Traditional Test Execution

Executing a runlist on your local system is quite simple. The `utah` command is the mechanism for doing this. Its most simple usage is done with:

```
# NOTE: the test runner needs root support things like "reboot" or
# allowing a testcase to install packages
sudo utah -r <runlist>
```

Where runlist can be a path to a local file or a BZR branch:

```
sudo utah -r /usr/share/utah/cliet/examples/pass.run
sudo utah -r lp:utah/utah/client/examples/pass.run
```

2.3.2 Touch Testing

The utah-client can be run directly on a touch device. If this is desired the previous section's instructions are correct. However, test development for Touch seems to now be more focused on writing tests that run on a host PC and then talk the target device over adb or phablet-tools. Its even possible to write tests that can detect which mode its being run under and then execute its steps accordingly.

Testing From Host

The run_utah_phablet.py script includes an option "–from-host" that basically launches utah on the host with ANDROID_SERIAL and TARGET_PREFIX exported into the tests environment. TARGET_PREFIX will be the path to the "adb-shell" command. While ANDROID_SERIAL is the serial of the device under test. A simple tc_control example that can work from host or target would look like:

```
description: uname shows an ubuntu-phablet image installed
dependencies: none
action: run uname
expected_results: command succeeds
type: userland
timeout: 60
command: $TARGET_PREFIX uname -a | grep ubuntu-phablet
```

The default touch tests should serve as an example of how to create a test suite that can run from host or target.

2.4 UTAH and autopilot

2.4.1 How do I run autopilot test cases through UTAH in a touch image?

To run `autopilot` test cases in a touch image through UTAH the following steps would be needed:

- Write UTAH a test suite that includes the autopilot test cases

  The easiest way to write the test suite, is to use a dynamic runlist as was described in the *tslist.auto* section. The reason for this is that it isn't needed to update the test suite if some test case is added/removed/renamed in the autopilot code. Thanks to the discovery command, there's no need to explicitly set the list of test cases.

  Note that a setup script should be added to install additional packages that might be required like, for example, `autopilot-touch`.

- Write UTAH a runlist that includes the test suite

  When writing a runlist, note that it's possible to let the touch device get the test suite using the `bzr-export` fetch method or do that outside of the device, push it to a `/tmp` location and let the device use the `dev` fetch method. Currently, the recommended option is the second one because it's more reliable.

Once the runlist is ready, the device has to be provisioned. This includes flashing the image, configuring the network and installing the UTAH client packages:

```
run_utah_phablet.py --skip-install --skip-network \
    -s <serial> --ppa ppa:utah/stable
```

After the device is provisioned, any files that are expected to be in the device like the ones for the test suite have to be pushed and the test case can be executed:

```
/usr/bin/run_utah_phablet.py --skip-install --skip-network --skip-utah \
    -s <serial> --results-dir <results_dir> --runlist <remote_runlist>
```

2.4.2 How does run_utah_phablet.py work?

Preparation

Before really running the `autopilot` test cases, `run_utah_phablet.py` performs the following actions:

- Write `/usr/local/bin/utah-autorun.sh` into the device. This script:
  - Looks into the working directory (`/etc/utah/autorun`) for executable files
  - Moves them one by one to `/var/cache/utah/autorun/inprogress`
  - Runs them
  - Moves them to `/var/cache/utah/autorun/complete`
  - Renames them to a suffix with their completion timestamp
- Write `/etc/init/run_utah.conf`. This is an upstart job description file that launches `/usr/local/bin/utah-autorun.sh` when the `run_utah` event is emitted and writes its output to `/tmp/utah.log`.
- Write `/usr/local/bin/utah-autopilot` into the device. This is a script that makes sure that permissions to `/dev/uinput` are correctly set and wraps `autopilot` to make sure it's called using the `phablet` user. To use this script, it must be explicitly called as a comment in the test suite as shown in the following `tslist.auto` file from the web browser application smoke tests:

```
-
  discovery_cmd: "autopilot list webbrowser_app | grep webbrowser_app | sed -e 's/^.*webbrowser_
  test_cmd: utah-autopilot run -v webbrowser_app.tests.{}
```

- Write a script to `/etc/utah/autorun/01_run-utah` that calls utah with the required runlist and enables/disables the shell if needed.
- Launch a process to capture `/var/log/syslog` and write it to stdout.
- Emit the `run_utah` event.

Run

Once the `run_utah` event is emitted, is when all the preparation work comes into play. Specifically:

- The upstart job defined in `/etc/init/run_utah.conf` is triggered.
- `/usr/local/bin/utah-autorun.sh` is executed and the output written to `/tmp/utah.log`.
- `/etc/utah/autorun/01_run-utah` is found in the working directory.
- `/etc/utah/autorun/01_run-utah` is moved to `/var/cache/utah/autorun/inprogress`.
- `/etc/utah/autorun/01_run-utah` is executed.
- `/usr/bin/utah` is launched with the runlist passed to `run_utah_phablet.py`.
- `/usr/local/bin/utah-autopilot` is called for each autopilot test case found by the discovery command.
- `/etc/utah/autorun/01_run-utah` is moved to `/var/cache/utah/autorun/complete`.

Results

Once the test cases have been executed, `run_utah_phablet.py` takes care of the following actions:

- Pull `/var/lib/utah/utah.yaml` from the device to the results directory.
- Pull `/tmp/utah.log` from the device to the results directory.
- Pull any other result files from the device that might have been requested.

2.5 Frequently Asked Questions

2.5.1 UTAH Client FAQ

How do I install the client?

```
sudo apt-add-repository ppa:utah/stable
sudo apt-get update
sudo apt-get install utah-client
```

Is there an example testsuite I can look at?

There is an example testsuite.

How do I run the utah-client locally?

```
sudo utah -r <master.run>
```

Why are there so many control files?

The goal of UTAH is to make sharing and reusing testsuites and testcases as easy as possible. The control files make it possible to define testcases and testsuites fully and allow a third party using a testsuite or testcase the control they need to run the them in their own environment and to suit their own needs.

Can I run the utah client on a local testsuite?

You can, but it must be in a supported (`bzr`, `git`) local repository.

Why doesn't the utah client support running a testsuite in the current directory or a local copy?

In order to maintain tracability utah needs to know which version of a testsuite was run. This is the reason we require testsuites be in a VCS. This makes troubleshooting test failures and reproducing test runs in other environments much easier and more reliable.

How do I get help with writing or migrating my tests to UTAH?

Send an email to ubuntu-utah-devel@lists.ubuntu.com and we will be happy to help you.

2.5.2 UTAH Server FAQ

What provisioning methods are currently supported?

Virtual machines via `kvm` and `qemu` are currently supported and physical machines via `cobbler`. The physical machine support is specific to the Ubuntu Engineering QA Lab setup.

Do you support provisioning LXC, existing VMS, of other methods?

Not at this time. There are plans for adding new provisioning methods and we'd be happy to review merge proposals. The branch is in launchpad under the UTAH project.

2.6 Provisioning

UTAH is very simple in terms of testing reliably and being able to reproduce a test run in a consistent manner. A command line specifies the test environment, type of machine to test on and a runlist. Then wait for it to finish and analyze the results.

UTAH can drive testing on whichever machine it is running or on a different one, by provisioning it with a particular provisioning method.

WARNING: Your machine will be modified, not to interfere with your current set-up or virtualized environment, so UTAH will create a new user called utah with sudo privileges to run the tests and have an inventory of existing VMs that you could be interested in for running tests.

2.6.1 Installation

There are some dependencies that need to be met in order to test with VMs, in a nutshell:

```
sudo apt-add-repository -y ppa:utah/stable
sudo apt-get update
sudo apt-get install utah
```

This will install the provisioning code as well as the test runner. To install just the test runner, install the utah-client package.

2.6.2 Provision and Test

Using a Touch Device

```
usage: run_utah_phablet.py [-h] [-s SERIAL] [--skip-install] [-r REVISION]
                           [--current] [--ubuntu-bootstrap] [--developer-mode]
                           [--channel CHANNEL] [--preserve] [--skip-network]
                           [-n NETWORK_FILE] [--skip-utah] [-b BRANCH]
                           [-e ENV] [--ppa PPA] [--results-dir RESULTS_DIR]
                           [--pull PULL] [-l RUNLIST] [-d DISCONNECTED]
                           [--device DEVICE] [--from-host] [--whoopsie]

Provisions a device for UTAH testing.

optional arguments:
  -h, --help            show this help message and exit
  -s SERIAL, --serial SERIAL
                        Android serial ID of device
  --skip-install        Skip running the phablet-flash install.
  -r REVISION, --revision REVISION
                        series/revision to install
  --current             Use the "current" image rather than "pending".
  --ubuntu-bootstrap    Use the system-images.ubuntu.com images.
  --developer-mode      Allow write access system-image partitions.
  --channel CHANNEL     Use an alternative channel for system-image.
  --preserve            Preserve the systems user data when flashing.
  --skip-network        Skip setting up wifi settings on device.
  -n NETWORK_FILE, --network-file NETWORK_FILE
                        Specify network manager config file for wifi. If none
                        is specified we'll try and use the default wifi
                        configuration found on host system.
  --skip-utah           Skip install the utah-client on device.
  -b BRANCH, --branch BRANCH
                        Install UTAH client on device from branch.
  -e ENV, --env ENV     Environment variables to set for utah-client.
  --ppa PPA             Specify an alternative PPA for utah.
                        default=ppa:utah/stable
  --results-dir RESULTS_DIR
                        Directory to store results in on the host.
                        default=/tmp
  --pull PULL           A file or directory to copy into the hosts's results-
                        dir after UTAH has been run. This option can be
                        specified multiple times.
  -l RUNLIST, --runlist RUNLIST
                        The utah runlist to execute
  -d DISCONNECTED, --disconnected DISCONNECTED
```

```
                        Uses the supplied script to disconnect USB while UTAH
                        is executed.
  --device DEVICE       The Android device type.
  --from-host           Executes from the host (the runlist must talk to the
                        target using ADB.
  --whoopsie            Run whoopsie-upload-all after the test to to ensure
                        complete .crash files are uploaded.
```

Using an x86 Virtual Machine

```
usage: run_utah_tests.py [-h] [-m MACHINETYPE] [-v VARIANT]
                         [--skip-provisioning] [-s SERIES] [-t TYPE] [-a ARCH]
                         [-n] [-d] [-j] [-f FILES] [-o OUTDIR] [--dumplogs]
                         [--outputpreseed] [-i IMAGE] [-p PRESEED] [-b BOOT]
                         [--rewrite {all,minimal,casperonly,none}] [-k KERNEL]
                         [-r INITRD] [--name NAME] [-e EMULATOR] [-x XML]
                         [-g GIGABYTES] [--diskbus DISKBUS] [-l LOGPATH]
                         runlist

Provision a machine and run a runlist there.

positional arguments:
  runlist               URLs of runlist files to run

optional arguments:
  -h, --help            show this help message and exit
  -m MACHINETYPE, --machinetype MACHINETYPE
                        Type of machine to provision (physical, virtual)
                        (Default is virtual)
  -v VARIANT, --variant VARIANT
                        Variant of architecture, i.e., armel, armhf
  --skip-provisioning   Reuse a system that is already provisioned (name
                        argument must be passed)
  -s SERIES, --series SERIES
                        Series to use for installation (lucid, precise,
                        quantal, raring, saucy, trusty, utopic) (Default is
                        trusty)
  -t TYPE, --type TYPE  Install type to use for installation (desktop, server,
                        mini, alternate) (Default is mini)
  -a ARCH, --arch ARCH  Architecture to use for installation (i386, amd64,
                        arm) (Default is amd64)
  -n, --no-destroy      Preserve VM after tests have run
  -d, --debug           Enable debug logging
  -j, --json            Enable json logging (default is YAML)
  -f FILES, --files FILES
                        File or directory to copy from test system
  -o OUTDIR, --outdir OUTDIR
                        Directory to store locally copied files (Default is
                        /var/log/utah/machine-name)
  --dumplogs            Write client output logs to standard out
  --outputpreseed       Copy preseed to logs directory and list as log file in
                        output
  -i IMAGE, --image IMAGE
                        Image/ISO file to use for installation
  -p PRESEED, --preseed PRESEED
                        Preseed file to use for installation
  -b BOOT, --boot BOOT  Boot arguments for initial installation
```

```
  --rewrite {all,minimal,casperonly,none}
                        Set level of automatic configuration rewriting
                        (Default is all)
  -k KERNEL, --kernel KERNEL
                        Kernel file to use for installation
  -r INITRD, --initrd INITRD
                        InitRD file to use for installation
  --name NAME           Name of machine to provision
  -e EMULATOR, --emulator EMULATOR
                        Emulator to use (kvm and qemu are supported, kvm will
                        be favored if available)
  -x XML, --xml XML     XML VM definition file (Default is /etc/utah/default-
                        vm.xml)
  -g GIGABYTES, --gigabytes GIGABYTES
                        Size in gigabytes of virtual disk, specify more than
                        once for multiple disks (Default is [8])
  --diskbus DISKBUS     Disk bus to use for customvm installation (virtio,
                        sata, ide) (Default is virtio)
  -l LOGPATH, --logpath LOGPATH
                        Directory used to write log files to

For example:
Provision a VM using a precise server image with i386 architecture and run the given runlist
    run_utah_tests.py -s precise -t server -a i386 \
            /usr/share/utah/client/examples/master.run
```

2.7 Developing UTAH

2.7.1 Provisionioning

Within the UTAH code, provisioning machines is handled by subclasses of the `Machine` class.

Scripts should provision machines by creating an instance of a subclass of the `Inventory` class, and using the `Inventory.request()` method to obtain a machine. Inventories are intended to prevent resource collision.

End users not writing scripts can provision machines automatically using scripts provided by the UTAH package. Please see the main UTAH page for that.

2.7.2 How to use custom preseeds

By default UTAH uses a preseed stored as `/etc/utah/default-preseed.cfg` for the provisioning of the test machine. By giving a **-p preseed_file** input to UTAH a custom provisioning could be carried out.

For example, the following command would provision the test machine with quantal server image with the options given in the contents of *preseed_file* and run the post installation tests given in *master.run*:

```
sudo -i -u utah /usr/share/utah/examples/run_utah_tests.py \
    -i http://cdimage.ubuntu.com/ubuntu-server/daily/current/quantal-server-i386.iso \
    -p /absolute_path_to/preseed-file /absolute_path_to/master.run
```

An example preseed for precise is given in the Ubuntu Installation Guide.

2.7.3 Coding guidelines for developers

Introduction

As UTAH codebase becomes more stable, there's an effort to make it not only work correctly, but also look correctly. To do that, we're using flake8, a static code analysis tool that is a combination of both pyflakes and pep8. This way, both syntax and PEP008 errors can be checked in a single shot.

What is PEP8?

PEP008 (Python Enhancement Proposal 8) is a document that contains a set of style guidelines for developers written by Guido van Rossum and Barry Warsaw. It's commonly accepted as the de facto style document for python developers; so, unless we don't find useful after some time, we decided to adopt it.

Are there any other guidelines?

Aside from PEP008, the Google Python Style Guide has received some attention as well. Since it differs from PEP008 at some places, this one isn't being followed at this time, but some part of it could be used in the future.

Are there any other tools?

Another tool that has a very good reputation is pylint. While it's a very useful tool, it's a little bit more difficult to use and requires some time to configure it to adapt to the policy of a given project. Anyway, it could be used in the future as well to catch some of the errors/warnings that might not be detected by flake8.

CHAPTER 3

Indices and tables

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