DART Automated Test Execution

User Manual

Lockheed Martin Advanced Technology Labs, May 18, 2015

ABSTRACT

This document provides a developer/tester with the knowledge to create and run automated tests against a diverse range of virtual (ESXi-based) and physical test resources.

Table of Contents

1 DART Automated Test Execution Technology Overview

1.1 What is Tyrant?

Tyrant is the name given by Lockheed Martin Advanced Technology Labs to a suite of technology it developed for running automated software tests. It is one of the major components of the complete DART (Dynamic Automated Range Testing) tool suite. (The other major component handles building out cyber test ranges.) The Tyrant suite allows multiple simultaneous developers and testers to run tests in a reproducible manner across a wide array of resources. The Tyrant suite also allows administrators to manage these test resources. Using Tyrant technologies, one can encode the logic of a test to perform against one or more test resources as well as the logic to determine whether the test is a success or failure. One can then run this test against a specific set of resources (i.e. the resources needed to run a single instance of the test), or have multiple instances of the test run against a diverse range of resources to evaluate the functionality of a system-under-test against the whole range of systems it may be run on in the real world. Finally, multiple developers can perform these tests simultaneously against a shared set of resources without conflicting with each other.

1.2 Technical Components Overview

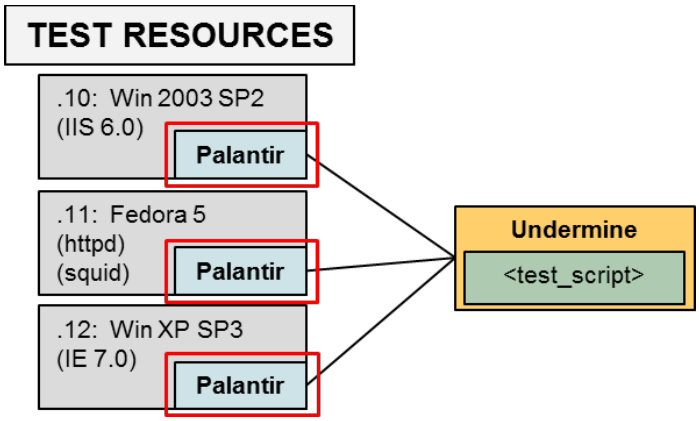
Tyrant is made up of the following central technical components:

- Palantir (Testing Interface)
- HAL (Hardware Abstraction Layer)
- Undermine (Test Script Harness) + Test Scripts
- Overmind (Test Script Scheduler) + Test Plans
- Overview (Test Resource/Result GUI)
- Reaper (Test Resource Sanitizer)
- Remote Commit (Remote Job Submission)

Each component is standalone from the others, allowing for each user to determine the combination of components most appropriate for their testing scenarios. The contents of this manual apply to DART Tyrant code released on May 6,2014.

1.2.1 Palantir

Palantir provides a common, cross-platform test interface to each of the computer resources in the test environment that are running commodity operating systems (e.g. Windows, Linux, OSX, FreeBSD, etc.). This allows test script writers to write more cross-platform test scripts, expecting a consistent interface (e.g. “put file”, “execute”, “spawn”, etc.) on each of the computers needed for the user’s test scenario.

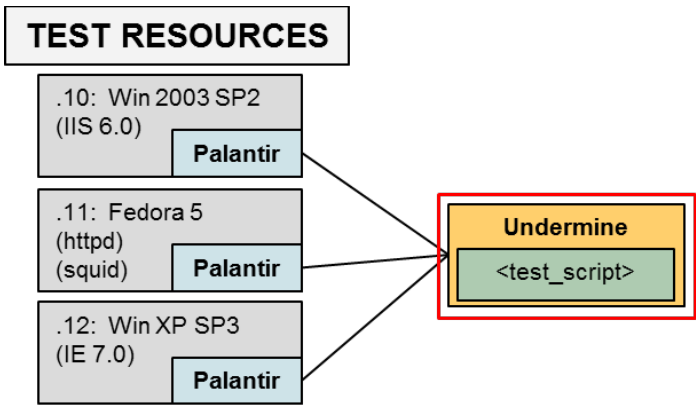


1.2.2 Hardware Abstraction Layer (HAL)

The HAL provides a framework for implementations of computer-level operations, i.e. operations done to the computer from outside of it, such as powering it on, powering it off or saving its state. These are all operations which can be performed without interacting with the operating system on the computer. Based on attributes of the computer being operated on, the HAL chooses the correct implementation of the logical operation desired (e.g. for a "power_off" operation, the HAL might use an ESXi control library for ESXi VMs, whereas for a physical computer, the HAL would use a configured PDU to power off the outlet the computer is connected to). For controlling saved states, various methods can be implemented as plugins to the HAL framework, such as reverting a VM snapshot for VMs, applying a disk image to physical machines, or using an auto-building system to automatically install an OS on a computer from scratch.

1.2.3 Undermine + Test Scripts

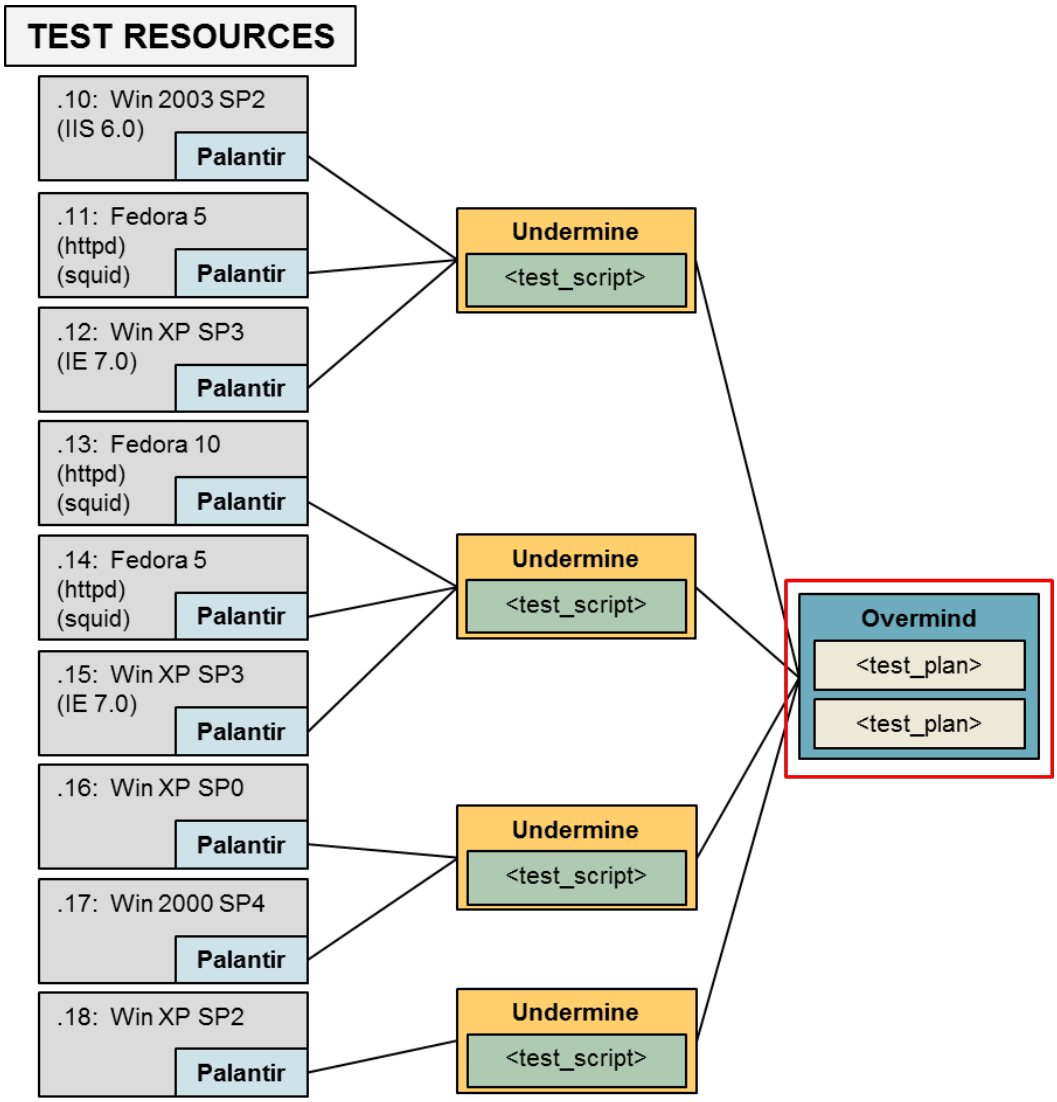
Undermine is a test script harness for executing the user's test scenarios. The user will encode their test procedure in a "test script". Then he/she will run that test script via Undermine, which will effectively automate their procedure. Undermine performs the actions on the test resources via Palantir if it exists on the test resource.



1.2.4 Overmind + Test Plans

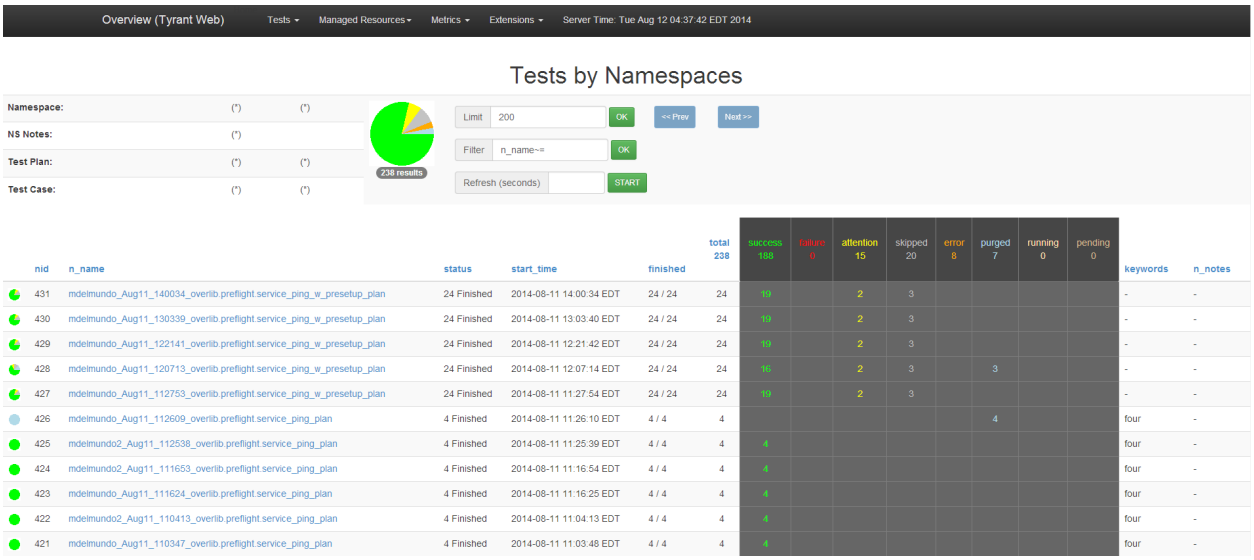
Overmind examines the set of test resources in the Test Resource section of the Overmind Database and schedules tests to be run across those resources. The user describes what tests they want to run, the

resource constraints, and the desired iteration and/or replication of each test script in a “test plan”. Multiple users can submit test plans concurrently, and Overmind will schedule each possible test to run when possible. Overmind runs an instance of Undermine for each test that needs to be performed. When tests are completed Overmind puts the test results into the Test Result section of the Overmind Database and marks the resource as requiring sanitization in the Test Resource section of the Overmind Database.



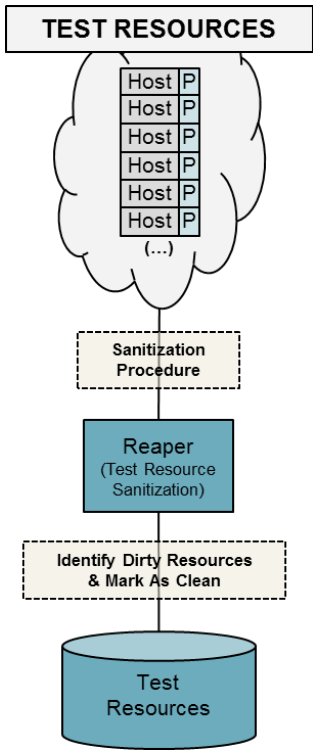
1.2.5 Overview

Overview is the web-based GUI that allows a user to view currently running and past test results in the Test Result section of the Overmind Database previously inserted by Overmind as well as graphically manage the resources in the Test Resource section of the Overmind Database.



1.2.6 Reaper

Reaper monitors the Test Resources section of the Overmind Database for resources that require sanitization and uses the HAL to restore a previously-saved, sanitized state to the computer. Different sanitization methods can be specified for each resource. After the sanitization process completes, Reaper marks the resource as clean in the Test Resource section of the Overmind Database, indicating to Overmind that the resource can be considered for use in a test again.



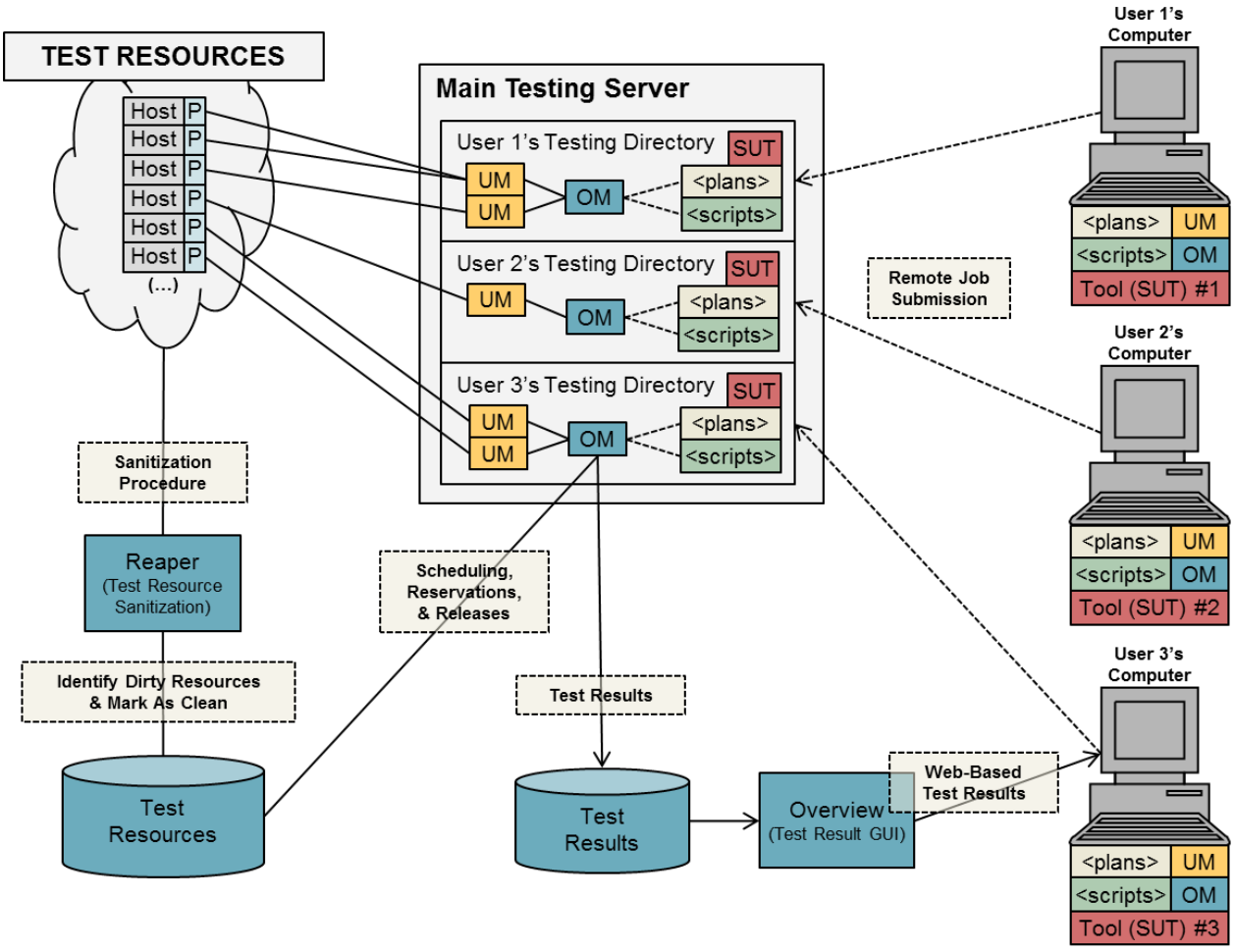
1.2.7 Remote Commit (Remote Job Submission)

While there are many ways to install and configure the Tyrant software, one very common setup is the one that allows for multiple users to submit and run their own tests on a set of common testing resources.

In this scenario each user has their own local copy of Overmind, Undermine, his/her tool or System Under Test (SUT), and test scripts and test plans relative to that SUT. Each user runs the `remote_commit` tool to copy those five components from their local box to the main testing server, start Overmind, and submit the necessary test plans. This ensures that each user can be running completely different tests from every other user. This also allows each user to continue their development of any of those five components without affecting currently running tests.

Each user can then browse to the Overview web GUI to monitor progress of their running tests.

The following picture illustrates all of the Tyrant components working together to execute this CONOP. ("P" == Palantir, "UM" == Undermine, "OM" == Overmind)



1.2.8 Plunger (Database Cleanup)

There are multiple ways to condense, delete and purge records from the Tyrant database. One approach is to run `bin/db_admin` commands such as `del_reservation_history`, `condense_contentions`, and `del_contentions`. Another is to run the extendable service, Plunger.

The administrator can determine what database condense and cleanup tasks to allow Plunger to run by setting options in the configuration file, *plunger.rc*. He or she can also add plunger modules in the `plunger_extensions` section of the configuration file to perform additional database maintenance functions.

1.3 For the Developer

For the developer, Tyrant provides the tools to create and run automated test scripts (in undermine) and test plans (in overmind), view test results, and work simultaneously on a range of resources. This manual covers how to use a range of physical and ESXi virtual machine resources previously set up by an administrator and how to create and run tests.

1.4 Repository Structure

The core of Tyrant is provided in two Mercurial repositories: tybase and tyworkflow. The tybase repository provides the tools used for running single tests. The tyworkflow repository provides the tools used for running tests across ranges of resources and managing these ranges. The tybase repository is standalone, but the tyworkflow repository has dependencies on tybase.

With the split of functionality between tybase and tyworkflow, it can sometimes be difficult to determine which repository you should be in to perform certain operations. Unless specifically directed otherwise by this manual, the following are a few general rules to help you determine the correct place to perform certain operations:

- If the operation relates to running an individual test against specific test resources without use of any range scheduling (e.g. overmind), you should be in tybase.
- If the operation relates to linking in collections of test scripts (leafbags), you should be in tybase.
- If the operation relates to scheduling tests to run on a range or managing a range of shared resources, you should be in tyworkflow.

The upsync_apps repository contains code used for testing with PSPs (Personal Security Products) and for automatic software updating. It also contains support for detecting PSP events by watching a resource's screen for changes.

The tyutils repository contains additional utilities used for various testing. Some utilities include support for usbs, window and controls, taking vmware screenshots, and event detection. For details about different utility usages, refer to their respective API documentation.

Related to tyutils are the PIL repositories (PIL-linux-i686 and PIL-linux-x86_64). These provide the Python Imaging Library, which is used by tyutils event detection to find changes between screenshots. Two exist because they contain compiled code which needs to match the architecture of the system it will run on.

In-depth descriptions of the contents of each repository are included in Appendix B.

1.5 Tools

This manual covers the following Tyrant tools, which will be used and configured by the range administrator (with the repository containing each tool in parentheses):

palantir (tybase): A component which runs on each test resource, allowing the resource to be controlled by test scripts via a TCP connection. Palantir serves on ports 51134 and 51135 (for Windows) and 51134 (for Linux).

comp_admin (tybase): A component which allows you to perform computer-level operations against a computer (e.g. `power_on`, `power_off`, `save_state`, etc.).

undermine (tybase): The component which runs a single instance of a test. Undermine connects via palantir to the resource(s) used by a test and runs the given test script.

create_resource (tyworkflow): A component which uses the HAL to save the current state of the given computer and then automatically creates a resource entry for that combination of computer and state in the database.

overmind (tyworkflow): The component which schedules tests to run simultaneously. Overmind uses a database of test resources and schedules these resources for incoming tests.

reaper (tyworkflow): The component which handles reverting a resource to a clean state prior to running a test. What this means depends upon the reaper module in use for a given resource. For a virtual machine, this usually means reverting to a snapshot, but for a physical computer, this could mean using some imaging or auto-building solution to build out a certain OS configuration on a resource on-demand for a test.

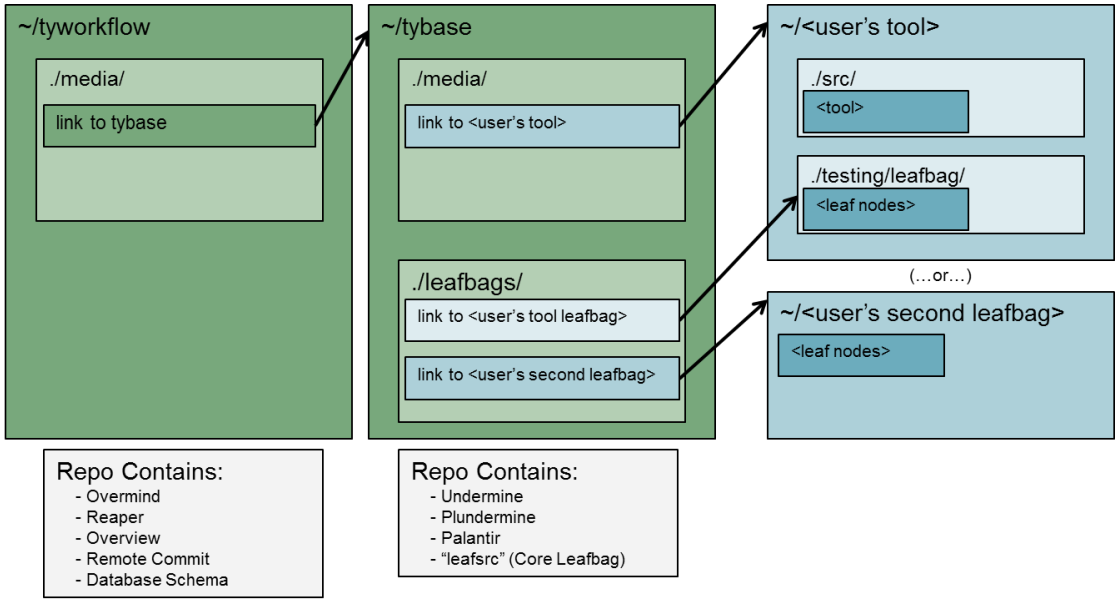
overview (tyworkflow): The component which displays test results and allows some management of test resources via a web-based interface.

process_plan (tyworkflow): A command-line tool used to process test plans (specifications for running multiple instances of tests against a specified variety of resources). This lets you submit plans to be run or see how many combos (specific runs of a test on specific resources and with specific parameters) would be run.

remote_commit (tyworkflow): A tool which allows remote submission of tests to a central Tyrant environment. While overmind on its own allows simultaneous tests to be run against a range of resources, remote commit makes overmind useful for a team of developers running different sets of code.

1.6 Directory Structure

Because each setup will involve testing different components, users will need to create a directory structure similar to the following:



Symlinks can be made between the various repositories using the standard `ln -s` command.

Additionally, a user should be in the base of either tyworkflow or tybase to run specific commands. For example, if a user was running `./bin/undermine`, he would be doing so out of the tybase directory. If he was running `./bin/remote_commit`, he would be doing so out of the tyworkflow directory.

1.7 Assumptions

This document makes some assumptions about the type of setup desired. Specifically:

- It is assumed that testing will be performed on VMWare ESXi virtual machines or physical machines connected to a network-addressable APC PDU.
- It is assumed that remote commit-style testing will be performed (with a team of developers, each at their own workstation, remotely submitting tests to a central server).
- It is assumed a mysql database will be used for storing resource and test information rather than sqlite3.
- The remote commit environment on the test server will run as root, and developers will submit their tests as root on the test server.
- The reader is familiar with working on the Linux command line and programming in Python.

2 Environment Setup

Before we can get to the work of writing and running tests, the environment must be set up. We will perform the setup now and verify its correctness in later sections.

On your developer workstation, clone the tybase and tyworkflow repositories into the same base directory. In the tyworkflow directory, run `make`. This links tyworkflow to tybase and unpacks our built-in python distribution.

A few configuration files need to be modified to enable you to use the range that's been set up for you. In tyworkflow, copy the file `rc/defaults/db.rc` to `rc/`. In the copied file, make the following changes (you may need to talk to the administrator to get this information):

- set `resource_manager/dbname` to the name of the overmind database
- set `resource_manager/engine` to mysql instead of sqlite3
- set `mysql/host` to the hostname or IP address of the test server, which is where the overmind database resides
- Set `mysql/user` and `mysql/passwd` to the username and password used to access the database you set in `resource_manager/dbname`. The default is "root", though Tyrant supports using a non-root database user. Your range administrator(s) may assign you a username and password to use, in which case you would set those values here.

Also, copy `rc/defaults/remote_commit.rc` to `rc/` and make the following changes:

- `remote_commit/remote_user`: The user to connect to the testing server as (via SSH when syncing up files or running commands. The default is "root", though Tyrant supports running remote_commit as a non-root user. Your range administrator(s) may assign you a username to use, in which case you would set that here.
- `remote_commit/remote_host`: the hostname or IP address of your testing server
- `remote_commit/remote_commit_dir`: the path on your testing server to the commits directory (`/proj/testing/commits` is the example used in the administrator manual).

For the above settings, make sure you uncomment any options which you set which are currently commented. That is, remove the semicolon from the beginning of any option line that you modify.

If you wish to change the default maximum number of post-test operation threads that may run concurrently, create `rc/overmind.rc`, and set the `max_popthreads` setting in the `overmind` section to the desired setting. For example, to set the limit to 10, you would put the following in `rc/overmind.rc`:

```
[overmind]
max_popthreads = 10
```

Post-test operations are operations which run in the context of the Overmind server after a test completes. An example of this would be automatically cloning resources used in a test that returned a certain result code. These operations run in threads in the server because they can have arbitrary

runtime, and therefore cannot reasonably be run in the server's main thread as part of normal test cleanup (since doing so would cause the server to hang while the potentially long-running post-test operations were executing). If you need to change this limit, you should consult with your range administrator to ensure this will not overload the range.

2.1 Viewing Resources

A simple task to start with is to view the list of resources. Navigating to the test server's `/overview` directory (e.g. http://testserver.example.com/overview) will present you with a menu of overview pages (Namespaces, Recipes, Resources, Management). If you click on Management, you should see a table listing all the resources you imported in the previous step.

2.2 Reserving Resources

When performing maintenance, one of the first things to do is reserve the computer you'll be working on. Reserving a computer prevents it from being scheduled for tests, so that it doesn't suddenly get used while you're working on it, and so that any changes you make to it while servicing it don't cause problems for developers' tests.

To reserve a computer in overview, navigate to the Management page (e.g. http://testserver.example.com/overview/add-computer.php). This page lists all the resources in the range and allows various tasks to performed on them. Check the box on the row for the computer you want to reserve, enter your name or some other sensible identifier in the "Reserver's Name" field on the left side of the page, and click the "Reserve" button. When the page refreshes, the resource you selected will have a yellow background for its row and the "use" field will say "N". To un-reserve a resource, making it available for testing again, check the resource's box and click the "Use Testing" button. The row will lose its yellow background and "use" will say "Y".

3 Leafnodes (Test Scripts)

Test scripts (known as "leafnodes" in Tyrant lingo) are the main units of work performed by undermine. Leafnodes typically involve one or more assets with palantir installed on them (but not always). Leafnodes can be as simple as a Python function that operates on some input parameters, to as complex as a class that choreographs a set of actions on multiple remote hosts via palantir. Where practical, leafnodes should be made smaller rather than larger to facilitate reuse.

3.1 Leafnode Concepts

Leafnodes come in different types (detailed below), but share some or all of the following concepts, which will be helpful to keep in mind for the rest of this manual:

- hosts: The test hosts, or resources against which the leafnode works. Some leafnodes do not use any resources, but most do, as running tests on computers is a primary purpose of leafnodes.
- inputs: If you think of the leafnode as a function, these are the arguments.

- progress messages: Asynchronous messages the leafnode can output during the course of its execution.
- result code: An indicator of the status of the leafnode returned when it's finished (e.g. success, failure, error).
- result (or output): A return value from the leafnode (different from its status).

3.2 Creating and Running a Simple Leafnode

We'll start by creating a simple leafnode, then having done that, delve into the details that will let you create more powerful test scripts.

To start, inside your tybase directory, create the directory path `leafbags/tutorial/tutorial` (the duplicate `tutorial` in the path is intentional). In this directory, create an empty `__init__.py` file to make it a valid python package.

Open up your text editor of choice and enter the following text:

```
from tybase.undermine.leaf import Leaf
from tybase.undermine.meta import leafi

@leafi.DefineActuator()
class Hello(Leaf):
    def run(self):
        host = self.hosts[0]
        ret = host.execcmd('echo', self.args[0])
        host.fwrite('/tmp/test.txt', self.args[0])
        dat = host.fread('/tmp/test.txt')
        if dat.strip() == ret.strip():
            return self.SUCCESS, 'cmd output matched file contents'
        else:
            return self.FAILURE, 'cmd output differs from file contents'
```

Save this as `leafbags/tutorial/tutorial/hello_world.py`.

In your web browser, navigate to the overview web interface running on the test server (e.g. http://testserver.example.com/overview). This is the interface by which you will later be able to view your test results, and will be covered in detail later. For now, click the Management link. Select a Linux resource whose "status" column says "avail", check the box next to it, put your name in the "Reserve Name" field on the left-hand side of the page, and click "Reserve". This reserves the machine for your use so that it won't be scheduled for others' automated tests, preventing your work from messing up others' or vice versa. Note the IP address of this resource.

In your tybase clone, run the following command, where IP_ADDR is the IP address of the resource you reserved:

```
bin/undermine tutorial.hello_world.Hello IP_ADDR -- "hello, world"
```

Here's an example of the output you should see:

```
    2013-09-23_16:51:02.44 (00191) [INF] script 22331: output_dir:
./output/undermine/nlsheppa/2013_09_23-16_51_02_348754
    2013-09-23_16:51:02.44 (00191) [INF] script 22331:  COMPLETION:
    success 'hello, world'
```

If you look on the actual Linux VM you reserved, you'll find a file `/tmp/hello.txt` with contents "hello, world".

You have written and run your first leafnode. Unreserve the resource you reserved previously by going back to the Management page, checking the box on the resource you reserved and clicking the "Use Testing" button.

3.3 Leafnodes in Depth

3.3.1 Writing Leafnodes

The general idea of writing a leafnode is that you write some sort of callable (see below) which receives zero or more palantir client objects, arguments and keyword arguments, performs some operations with them, and then returns a result code (see below) and result value, optionally with some progress message along the way. The callable is decorated with various decorators to define "metadata" for the leafnode.

In this section, we first illustrate the different ways of writing leafnodes, then dive in to the details of how to define metadata on them.

3.3.1.1 Types of Leafnodes

Here we cover the two main styles of leafnodes, class-based and function-based.

3.3.1.1.1 Type 1: class-based leafnodes

Class-based leafnodes are the most powerful and consist of a class which inherits from the Leaf class provided in tybase (by importing tybase.undermine.leaf.Leaf) and overrides certain methods (all of which take no arguments other than the `self` reference to the instance they're bound to).

In this style of leafnode, `self` is your reference to the currently running script. You access your host(s) through `self.hosts`, which is a list of palantir client objects (even if the leafnode only takes one host). Your arguments are provided initially in `self.args` (positionally-specified arguments) and `self.kwargs` (arguments specified with keywords) without regard to what's defined in the input parameters. You'll probably want to normalize your args by calling either `self.normalize_args` or `self.normalize_kwargs` in your run method (see below).

The methods which the leafnode may override are:

- `runSetup`: Run before the body of the leafnode. If this raises an exception, the leafnode will stop and the SKIPPED result code will be returned.

- `run`: The body of the leafnode. If this raises an exception, the leafnode will stop and the ERROR result code will be returned. Otherwise, this method must return a tuple of the result code and output value of the leafnode.
- `runCleanup`: Run after the body of the leafnode in all cases except when the leafnode times out, regardless of the leafnode's result code. If `runSetup` has an error other than a timeout, `runCleanup` is still run (so `runCleanup` is similar to the finally block of `try ... except ... finally`). The success or failure of `runCleanup` does not affect the final result code of the leafnode. If the leafnode body returns SUCCESS, the final result code will be SUCCESS even if `runCleanup` throws an exception.
- `stopHandler`: Run in the case of a timeout, when the leafnode is being stopped. This method has a limited time (currently five minutes) in which to run, which is why it's separate from `runCleanup`, which doesn't have as small a time limit.
- `hangDetectedHandler`: Run whenever one or more assets involved in the test appear to be hung at the end of the test (only if hang detection is enabled via the "-H" switch to undermine).
    - An asset is considered to be possibly hung if it fails to respond to a palantir ping. Thus, hang detection will catch a host which has a bluescreen or kernel panic, but will also regard a host whose palantir service has crashed or who has lost network connectivity as being hung. The `hangDetectedHandler` method receives as a parameter a list of the indices into the self.hosts list of the assets which appear to be hung. You can then use this information to perform whatever operations you wish on the hung hosts (or, technically, any of the hosts in the test).
    - An example `hangDetectedHandler` implementation is provided in the tyutils repository, in the module `leafbag/tyutils/leaf_fetch_dump_on_hang.py`. This module defines a `Leaf` child class (`LeafFetchDumpOnHang`) with a `hangDetectedHandler` implementation which reboots each hung machine and then, if it is a Windows machine, attempts to fetch a memory dump from the machine and store it in the test output directory. Since `LeafFetchDumpOnHang` is a child of the standard Leaf class, you can write your leafnodes to inherit from it instead of Leaf in order to include the memory dump fetching handler in your leafnodes.

Here's an example python script illustrating this type of leafnode (the DefineActuator and other metadata decorators will be covered later):

```
from tybase.undermine.leaf import Leaf
from tybase.undermine.meta import leafi

@leafi.DefineActuator()
class MyLeafnode(Leaf):
    def runSetup(self):
        #here you do setup tasks, like perhaps installing some
        #supporting piece of software on an asset
        pass
```

```
def runCleanup(self):
    #here you do cleanup, like perhaps deleting some temporary
    #files
    pass

def run(self):
    #the body of your leafnode

    #normalize to a dict of kwargs so we get our default values

    #and everything easily accessible by name, even if the args
    #were given positionally
    self.kwargs = self.normalize_kwargs()

    #a common thing to do if you only have one host
    host = self.hosts[0]

    #do some testing stuff
    #perhaps we want to measure how much data was exchanged over
    #the network in this case, our result value is an integer;
    #data type specification will be explained further on
    traffic_size = some_measurement_function()

    return (self.SUCCESS, traffic_size)

def hangDetectedHandler(self, hung_host_nums):
    #here you attempt to handle hung hosts
    pass
```

3.3.1.1.2 Type 2: function-based leafnodes

When you have a simpler testing task, you might choose this second type, in which you simply write a python function, which you optionally decorate with some metadata. This type of leafnode is simpler, but also less powerful. Differences compared to class-based leafnodes are:

- Your reference to the currently running script is stored at the `context` attribute of a host object. This unfortunately means that if your function-based leafnode takes no hosts, you will not have access to a reference to the currently running script and will not be able to do things which require it, like running a sub leafnode.
- You cannot define setup and cleanup logic like you can with runSetup and runCleanup for class-based leafnodes.
- You have less flexibility in setting the result code of your leafnode, as follows:
    - If your function raises an exception, ERROR will be returned, with the exception as the result value.
    - If your function returns at all (whether True, False, a number, None, anything), then SUCCESS will be used.

- Input parameters are typically defined implicitly by the function prototype, rather than explicitly with metadata on the leafnode.
- Arguments are handled differently. The arguments that are passed to your function-based leafnode consist of the host objects, followed by the args, followed by the kwargs. Thus, if your leafnode is called with too many or two few hosts, you can end up with an argument you expected to be a host containing an arg value (too few hosts), or an arg containing a host rather than the arg value you expected (too many hosts).
- Hosts, args and kwargs are accessed by the names you give them in the function prototype, as with any function.

Here's an example python script illustrating a function-based leafnode:

```
import tybase.undermine.meta.leafi as leafi

def my_leafnode(host1, host2, arg1, arg2, kwarg1=0, kwarg2=True):
    #do some testing stuff
    #no matter what you return here, the result will be SUCCESS (as
    #long as you actually return and don't throw an exception)
    return True
```

3.3.1.2 Leafnode Metadata

Leafnode metadata is how you define things like what kinds of assets your leafnode works against, what kinds of data it takes on input and output, whether it provides asset properties, etc. Metadata is set on a leafnode by decorating the leafnode with decorators provided in the tybase.undermine.meta.leafi module. Metadata is not strictly required to run a leafnode, but it is advised, and it is necessary to take advantage of certain advanced leafnode features. This section's subsections explain the provided metadata decorators organized by the purpose they serve, with the name of the decorator in parentheses.

3.3.1.2.1 Defining the Leafnode Purpose (DefineActuator, DefineSensor, DefineProcessor)

These three mutually exclusive decorators describe the function the leafnode serves and how it will interact (or not) with any assets it uses. Currently the usage of these decorators is only by convention; they don't do anything special to the leafnode you put them on (except for DefineSensor with asset properties). However, leafnodes need to have some type of metadata, and putting one of these on the leafnode is a good way to satisfy that requirement.

The convention for these decorators is:

- DefineActuator: leafnodes that make changes to an asset (e.g. delete a file, install a piece of software, etc)
- DefineSensor: leafnodes that only query information on an asset, not make changes to it. If you want your leafnode to assert (provide) asset properties, it must have this decorator.
- DefineProcessor: leafnodes that do not care about what assets they receive, and may not even take any hosts at all

3.3.1.2.2 Defining Input Parameters (Inputs)

This decorator takes as its arguments tuples defining

- the name of an input parameter
- its data type (see below for valid types and how to specify them)
- optionally a default value for the parameter.

This decorator only makes sense for the first type of leafnode (class-based), since the other two types depend on the arguments defined in the function prototype.

An example usage is:

```
import tybase.undermine.meta.leafi as leafi

@leafi.Inputs(
    ('num_runs', int),
    ('interval', float, 5.0),
    ('path', str, 'C:\\test_dir'),
    ('quick_run', bool, False)
)
class MyLeafnode...
```

3.3.1.2.2.1 Input and Output Data Types

Leaf node parameter data types can be any scalar type that can be pickled, as well as lists or structs.

For scalar types, the data type definition (the second field of the input parameter tuple, or the argument to the FinalOutput decorator) is simply the type. For example:

> `@leafi.Inputs(('num_runs', int, 5))` (defines a single input parameter of type int named num_runs with default value 5)
>
> `@leafi.FinalOutput(bool)` (defines a result value of type bool)

Typical scalar types are str, int, float and bool.

For complex types, you show the data types of the scalar parts of the complex types in the context of that type (as a list for lists, as a dict for structs). Also, complex types may be nested. For example:

> `@leafi.Inputs(('animals', [str]))` (defines a single input parameter which will be a list of strings named animals and have no default value)
>
> `@leafi.FinalOutput({'MemTotal': int, 'MemFree': int, 'Swapfile': str})` (defines a result value which will be a struct with three fields of type int, int and str, respectively)
>
> `@leafi.FinalOutput([{'name': str, 'lat': float, 'long': float}])` (defines a result value which will be a list of structs, each representing a city)

`@leafi.FinalOutput({'size': int, 'files': [str]})` (defines a result value which will be a struct of a size value and a list of filenames [perhaps this is the size and contents of some archive file the leafnode processed])

However, valid leaf node lists and structs are more limited than what can be expressed in Python lists and dicts, as follows:

3.3.1.2.2.1.1 Lists

Lists are defined by giving the type of the scalar values of the list in list context in the input/output definition, as seen in the above table. Thus, every element contained in a list must be of the same type. If you need to pass a static set of values of different types, consider using a struct instead. If you really need to pass a variable number of items of different types, consider (1) a list of structs, or (2) multiple lists, each of which contains a different type.

3.3.1.2.2.1.2 Structs

Structs are defined by giving a dict whose keys are the names of the struct fields and whose values are the scalar types for each field. The difference between the leafnode struct type and regular Python dicts is that structs are defined with a static set of fields. If you really need to input (or output) a dict-like data structure, here are a couple of options:

```
('keyVals1', [[str]]), # only if key and value types are the same

('keyVals2', [{'key':str, 'val':int}]) # different types for key & value
```

Even with the above alternatives, leafnodes are still restricted in that they cannot input or output arbitrary types.

3.3.1.2.3 Deriving Inputs from Function Introspection (DeriveInputs)

For function- and method-based leafnodes, where input parameters are based on the function/method definition, this decorator will use function introspection to automatically generate input parameter metadata (as you would specify with the Inputs decorator for class-based leafnodes) from the function/method definition. You simply provide this decorator with no arguments, like so:

```
@leafi.DeriveInputs()
def my_leafnode(...)
```

3.3.1.2.4 Defining the Result Data Type (FinalOutput)

The data type of the result value, aka output, is defined using the FinalOutput decorator, which takes the data type specification as its argument. This decorator uses the same specification as the Input decorator, as explained above.

3.3.1.2.5 Defining the Progress Message Data Type (ProgressOutput)

The progress message data type is defined just as with the result data type, but using the ProgressOutput decorator instead of the FinalOutput decorator.

3.3.1.2.6 Defining Leafnode Alias (Alias inside of a Define*)

The Define* decorators do have another purpose. Inside of a Define* decorator, you can also specify an alias with the Alias class provided in the leafi module. This is used with polymorphic leafnodes. You define an alias like so:

```
@leafi.DefineActuator(leafi.Alias('uber_leafnode'))
def my_leafnode(...)
```

3.3.1.2.7 Defining the Leafnode "Subjects" (assets to run against) (Subject)

Using the Subject decorator, you can define constraints to restrict what your leafnode can run on. These constraints utilize asset properties (a list of which can be found here). In the Subject decorator, with the constraints keyword, you specify a list of constraint comparisons with the Prop, AndProp and OrProp classes provided in the leafi module. All the elements of the constraints list must match for the leafnode to be allowed to run.

Some examples:

Require 64-bit Windows 7:

```
@leafi.Subject(
    constraints=(
        leafi.Prop('sw.os.architecture') == 'x86_64',
        leafi.Prop('sw.os.name') == '6.1',
        leafi.Prop('sw.os.family') == 'Windows'
    )
)
```

Require 64-bit Windows 7 (illustrating the use of AndProp, which is unnecessary, but valid):

```
@leafi.Subject(
    constraints=(
        leafi.AndProp(
            leafi.Prop('sw.os.architecture') == 'x86_64',
            leafi.Prop('sw.os.name') == '6.1',
            leafi.Prop('sw.os.family') == 'Windows'
        ),
    )
)
```

Require 32-bit Windows 7 or XP (illustrating the use of OrProp):

```
@leafi.Subject(
    constraints=(
        leafi.OrProp(
            leafi.Prop('sw.os.name') == '5.1',
            leafi.Prop('sw.os.name') == '6.1'
        ),
```

```
        leafi.Prop('sw.os.architecture') == 'x86_64',
    )
)
```

3.3.1.2.8 Defining the Default Leafnode in a Module (MainLeaf)

If you so choose, you can mark a leafnode in a module as the default leafnode for that module. Then, when you specify the leafnode to run (e.g. on the undermine command line), you need only specify as far as the module, and undermine will automatically run the default leafnode you marked. To do this, place the MainLeaf decorator on the desired leafnode.

For example, consider the example leafnode we created and ran in Creating and Running a Simple Leafnode. If we added the MainLeaf decorator to the Hello class, then rather than referencing the leafnode as `tutorial.hello_world.Hello` like before, you could reference it as simply `tutorial.hello_world`.

The MainLeaf decorator would be added to the example leafnode like so:

```
@leafi.DefineActuator()
@leafi.MainLeaf()
class Hello(Leaf):
```

3.3.1.2.9 Inheriting Metadata from Parent Classes (InheritMeta)

With the class-based style, where class inheritance is involved, this decorator can be used to inherit metadata defined on a parent class to the child class. You put this decorator on the class and the class would inherit metadata from parent classes.

3.3.1.3 Inside the Leafnode

At this point, we know how to structure a leafnode, but what goes in the body?

Technically, pretty much whatever you want within the limits of python. Typically, you interact with palantir client objects to put or get files from assets, run commands on them, do operations influenced by the input parameters, and so forth. To see a list of the operations available with palantir, run `bin/palantir_admin -h` in your tybase clone. Each of the methods documented therein are accessible on the host objects. For example, to put a file, you'd do:

```
self.hosts[0].put(src, dest)
```

Also, computer-level operations may be performed inside test scripts via the `hal` attribute of the palantir client object. This `hal` attribute has methods identical to the subcommands available with tybase's `comp_admin` tool (run `bin/comp_admin -h` in your tybase clone to see a list of available operations). For example, if you wanted to hard power cycle a host during a test (e.g. if you detected the host's OS had crashed), you could use the following line of code:

```
self.hosts[0].hal.power_cycle()
```

However, be advised of the following caveats when using computer-level operations:

- If you use the restore_state operation on a VM whose snapshot name is set to "latest" in the database in order to change which ESXi snapshot the VM is running, then all future tests will use

that changed snapshot (since the “latest” setting in the database means to simply restore the current snapshot, whichever one the VM happens to be on).
- State-related commands for machines using the clonezilla state control implementation only work when run on the test server, so test scripts which depend upon state control with clonezilla must be run through `remote_commit` or run from the main test server.

During the leafnode, you may choose to emit progress messages and at the end you return a result code and a result value (depending upon leafnode style).

3.3.1.3.1 Emitting Progress Messages

To emit a progress message, you call the `emitProgress` method on the Leaf class. If you're in a class-based leafnode, that means calling `self.emitProgress`. If you're in a function- or method-based leafnode, you need a reference to the currently running script, which is available on your palantir client objects as the context attribute (e.g. `host.context.emitProgress`). This method is defined as follows:

```
def emitProgress(self, data, seq=-1, tstamp=None, spath=None, dpath=None)
```

where the parameters are:

- `data`: The value of the progress message, which must match the type you defined in the ProgressOutput decorator.
- `seq`: Sequence number of the progress message (defaults to one greater than the last one, starting with zero)
- `tstamp`: Time stamp of the progress message, defaults to the current time.
- `spath`: Perhaps this means source path, but it appears to have no meaning and is rarely used, if at all.
- `dpath`: Override the path to the file where the progress message is stored, defaults to a file in the output directory whose name includes the sequence number. This is rarely used.

Usually, you should only provide data. An example call would look like:

```
self.emitProgress({'currentSpeedMPH':88.8})
```

3.3.1.3.2 Returning a Result

When in a class-based leafnode, you return a tuple of the result code and the result value. The result codes are constants defined on the Leaf class, as enumerated below. The result value is whatever value you want, of the type you defined in your FinalOutput decorator.

When in a function-based leafnode, as explained above, you simply return the result value and the result code is determined for you.

3.3.1.3.3 Leafnode Result Codes

The valid leafnode result codes are defined as attributes on the Leaf class and are as follows:

- SUCCESS: test completed without error and returned desired results (e.g. your software works)
- FAILURE: test completed without error and returned incorrect results (e.g. your software doesn't work, but your test is written properly)

- ATTENTION: test completed without error and returned generally desired results, but something is fishy and you want a human to follow up
- SKIPPED: test had an error in setup (either in basic undermine stuff like reaping an asset or in your optionally provided setup code)
- ERROR: test had an error while running the body of the test (e.g. your test has a problem and threw an exception)

Based on the convention of these result codes, your code should generally only explicitly return SUCCESS, FAILURE, or ATTENTION. If you want ERROR, you should throw an exception describing the error. Your leafnode body should not return SKIPPED since that is for setup problems.

3.3.1.3.4 Normalizing Arguments

Note: This only applies to class-based leafnodes.

By default, the code that calls your leafnode's functions does nothing to set up the arguments for you. That is, when an instance of your leafnode is created, you get some positional arguments and some keyword arguments based on how your leafnode was called. Another effect of this is that default values are not handled at all, which means you end up with `None` for the value of any arguments which are not provided in the call to your leafnode. So that you don't have to write your own argument handling code, two methods (`normalize_args` and `normalize_kwargs`) have been provided. These are methods on the Leaf class, so you can access them simply by calling `self.normalize_args` or `self.normalize_kwargs` from your class-based leafnode. Both take no arguments.

`normalize_args` returns the arguments to your leafnode as a list of positional arguments. `normalize_kwargs` returns the arguments as a dict keyed by input parameter name. Both set default values for arguments which are not provided (if you gave default values in your leafnode's metadata) or otherwise throw exceptions (if you did not give default values). Also, you may only call one of these functions (if you call both, then you may get errors about arguments being provided multiple times). The idea is for you to set `self.args` or `self.kwargs` to the return of the respective normalize function (e.g. `self.args = self.normalize_args()`). If your leafnode overrides the `__init__` method, that would be a good place to put it, otherwise you could just put it near the beginning of your run method.

3.3.1.3.5 Logging Information

Logging information may be output from a leafnode using the `log` attributes present on the script class and each host object. These `log` attributes are instances of the python logging module's Logger object (see http://docs.python.org/2/library/logging.html for full details on using python logging). Logging entries output using a host object's `log` attribute are tagged with the IP address or hostname of that host. All these log entries will show up in the script.log file in the leafnode instance's output directory (explained further in the "Running Leafnodes" section).

Examples:

- Logging an informational message to the script class's logger in a class-based leafnode:
  `self.log.info("something happened")`
- Logging an informational message to the script class's logger in a function-based leafnode (assuming the leafnode takes at least one host and the first parameter is named "host"):
  `host.context.log.info("something happened")`
- Logging a warning related to a specific host in a class-based leafnode:
  `self.hosts[2].log.warning("something might be wrong")`

3.3.1.3.6 Performing Palantir Operations as a Normal User (Windows Only)

Normally, when you perform operations on a test resource with palantir (i.e. using methods on one of your host objects), that operation is run on the test resource by a palantir processing running as the SYSTEM user. If you have an operation that you need to run as a regular user (e.g. because you need to interact with the GUI on modern Windows or need the operation to run with limited user privileges), this can be done using a system known as emissary which is built into the tybase repository.

In the body of your leafnode, add a line like the following (this assumes that "host" is the palantir host object on which you would run other test operations such as host.put, host.execcmd, etc):

```
      emhost = host.createEmissary(domain='DOMAIN',
username='USERNAME',
      password='PASSWORD')
```

Replace `DOMAIN`, `USERNAME`, and `PASSWORD` with the domain name, username, and password of the account you want to run as. If the specified user is not logged in, auto-login registry keys will be set and the test resource will be rebooted to cause the desired user to log in. If domain, username, and password are all omitted, then operations will be run as whatever user is currently logged in. If only domain is omitted, then the domain will be ignored when determining whether the correct user is logged in and when specifying via the registry what user to automatically log in.

Once this line of code has run, `emhost` will be a reference to an instance of palantir running as the specified user. This works exactly the same as any other palantir client object; it has exactly the same methods and properties, the only difference is what user the remote palantir server is running as.

3.3.1.3.7 Dropping to the Python Debugger

Sometimes when writing a new leafnode, it may be helpful to use the Python Debugger (pdb) in order to debug issues with the leafnode. Since the leafnode is just Python code, you can use pdb within it like you would in any other Python script.

NOTE: This method is only useful when running leafnodes one-off through undermine, since you need to be able to interact with the input and output of the leafnode in a terminal. If you attempt to use this when running leafnodes through overmind (or even plundermine), your leafnodes will hang at the point when pdb is invoked (because the script will be at a pdb prompt waiting for your input, but there will be no way for you to provide it).

To drop to a pdb prompt, insert the following line of code into your leafnode:

```
import pdb; pdb.set_trace()
```

The leafnode will drop to the pdb prompt at whatever point you insert this code. Some useful locations to put this pseudo-breakpoint are:

- At the top of the `run()` method, allowing you to set real pdb breakpoints and then continue execution of the script.
- Inside an except block that pdb runs only when some error you're trying to debug occurs.
- Inside a conditional block that detects some interesting condition you wish to investigate.

A full discussion of how pdb works is beyond the scope of this document. However, one useful pdb feature is the ability to skip tracing into certain libraries. For example, you may have an issue in your leafnode which you are sure is in your leafnode itself and not in provided tyrant libraries. In this case, it would be a waste of time to trace through execution of, say, the palantir communication libraries (which are usually called very frequently). Pdb provides the ability to specify certain module name patterns which should be skipped. To use this feature, replace the previously-provided line of code with the following:

```
import pdb; pdb.Pdb(skip=MODULE_PATTERNS).set_trace()
```

where MODULE_PATTERNS is an iterable of glob-style patterns of module names to skip. For example, to skip tracing the entire tybase library, use:

```
import pdb; pdb.Pdb(skip=['tybase.*']).set_trace()
```

To skip only the palantir communication code, use:

```
import pdb; pdb.Pdb(
      skip=['tybase.palantir.*','tybase.support.netcom.*']).set_trace()
```

For full details about pdb, see http://docs.python.org/2/library/pdb.html.

3.3.1.3.8 Preventing Get File Collisions in Overmind Tests

Often, a test would need to get files from a test resource and put it on the Tyrant server for post-processing or viewing.

When getting files from the remote resource, using the get() function, if the local file argument is not specified, the file will be written to the current working directory of the undermine process running the test. This behavior could cause problems when running the test in Overmind since multiple instances of undermine will be running using the same current working directory. Files of the same name would be written to the same directory, thus overwriting previously-existing files of the same name from other tests. To prevent this, specify the test output directory path for the destination of the file. Correct example:

```
host_file = "test.txt"
host.get("c:\\" + host_file, os.path.join(self.output_dir, host_file))
```

3.3.2 Storing Leafnodes (Modules and Leafbags)

Leafnodes are stored in python modules in what's known as "leafbags". A leafbag has a very specific definition which must be followed: A leafbag is a directory which contains python packages. These python packages then contain python modules with leafnodes in them.

When undermine runs a leafnode, the roots of all the configured leafbags are on the python path. This is why, when running a leafnode, you can specify it as a "python import path", like you were trying to import it in a python script. Deep down in the guts of undermine, that is what is actually happening.

3.3.2.1 Structuring Leafnode Modules

Leafnodes are stored in python modules containing one or more leafnodes and having one of several markers within the first 200 bytes of the file. When a module has one of these markers, we call it "leafy". A module must be leafy in order for it to be scanned when tybase's `bin/prepare` is run. This marker allows the leafnode scanner to quickly ignore modules that have a very low potential of containing leafnodes.

Valid leafy markers are as follows:

- the string "THIS_IS_A_LEAF_MODULE"
- the string "#AUTOGENERATED" at the beginning of a line
- an import involving tybase.undermine.meta.leafi, e.g. `from tybase.undermine.meta.leafi import foo` or `import tybase.undermine.meta.leafi`
- an import involving tybase.undermine.leaf
- an import involving tybase.undermine.main_script

3.3.2.2 Structuring Leafbags

Your leafbag should have one or more levels of subdirectories and you should put leafnodes in these subdirectories (not in the root of the leafbag). In general, you should try to create your leafbags to either be entirely self-contained, or to depend on other leafbags (which you would then link in like normal, with no added complexity). This makes things easier in the long run. However, this is not always practical. If your leafbag depends on other files from elsewhere in your project's repository or just on other files in general, you will need to be aware of this and take it into account when using remote commit. See the section on leafbags with non-leafbag dependencies for how to handle this.

Since the directories in your leafbag are being treated as python packages, they must have `__init__.py` files like any python package. When you run `bin/prepare`, a component of tyrant will scan the leafbag. The leafbag scanner will only recurse into a subdirectory of the leafbag if at least one of the two following conditions is true:

- the subdirectory's `__init__.py` file contains the leafbag marker (the comment #LEAFBAG)
- the `__init__.py` file in an ancestor of the subdirectory up to but NOT including the root of the leafbag contains the leafbag marker with the RECURSE flag (the comment #LEAFBAG RECURSE). The RECURSE flag tells the scanner to go through all the subdirectories regardless of whether they have a leafbag marker.

So, the simplest thing is to just put an `__init__.py` in each top-level subdirectory of your leafbag with the comment #LEAFBAG RECURSE. If for some reason you have directories in your leafbag

devoid of leafnodes (such as a large third-party python module), then you might choose to not use recursion globally and only put the leafbag marker in specific subdirectories' `__init__.py` files.

3.3.2.2.1 Leafbag structure example

Consider an example software project (call it eproj) whose developers want to perform automated testing with leafnodes. This project may have a code repository (also named eproj) with a subdirectory called tests which is the leafbag for this project. With this in mind, consider the following partial directory and file structure for the example leafbag:

- `eproj/` (root of the repository)
    - `tests/` (root of the leafbag)
        - `utils/` (supporting python modules, not leafnodes)
            - `__init__.py` (has no leafbag marker)
        - `net_tests/` (leafnodes for testing the software in a network)
            - `__init__.py` (contains leafbag marker)
            - `test_data/` (some kind of data used for the tests and some python modules to work with it, but no leafnodes)
                - `__init__.py` (exists to make this a valid python package, but has no leafbag marker)
        - `standalone_tests/` (leafnodes for testing the software on a single computer)
            - `__init__.py` (contains leafbag marker with RECURSE flag)
            - `win_xp/` (leafnodes for Windows XP)
                - __init__.py (no leafbag marker)
            - `win_7/` (leafnodes for Windows 7)
                - __init__.py (no leafbag marker)
            - `notes/` (contains no `__init__.py` at all)

With this structure, the scanner will

- skip utils/ (since it has no leafbag marker, and it's a top-level subdirectory so there is no chance of having an ancestor with a leafbag marker with the RECURSE flag)
- look in net_tests (since it has a leafbag marker in its `__init__.py`)
- skip net_tests/test_data (since it doesn't have a leafbag marker and no ancestor has the RECURSE flag)
- look in standalone_tests and any subdirectories (since the `__init__.py` in standalone_tests has a leafbag marker with the RECURSE flag), BUT…
- skip standalone_tests/notes (since it has no `__init__.py` at all and is therefore not a valid python package)

3.3.2.3 Linking-in leafbags

In order to use leafnodes in a leafbag with tyrant, you must link this leafbag in to your tyrant repo. The typical way to do this is to create a symlink in the leafbags directory of tybase that points to the leafbag you want to use. An alternative is to actually put your leafbag in the leafbags directory of tybase, but this

usually doesn't make sense from an organizational standpoint because your typical project using tyrant has its own repository with the leafbag as a subdirectory.

So, following the example leafbag above, assuming your current working directory is the root of tybase and that the eproj repo is checked out in the same directory as tyrant-dev, you would run the following command to link in the leafbag:

```
ln -s ../../eproj/test leafbags/eproj
```

This results in a symlink called eproj in leafbags that points to the leafbag in the eproj repo.

3.3.2.3.1 Mitigation of naming conflicts

Note that leafbags can cause naming conflicts. For example, consider two projects, eproj and fproj. Suppose that both projects have leafbags with subdirectories named net_tests. In this case, if both leafbags are linked in to tyrant at the same time, a naming conflict will occur. The preferred way to mitigate this is to add an extra directory level in the leafbags named for the project. For example, in the current situation, the `net_tests` subdirectories that are conflicting are located at `eproj/tests/net_tests` and `fproj/tests/net_tests`. To mitigate this, one could add an extra directory level to end up with `eproj/tests/eproj/net_tests` and `fproj/tests/fproj/net_tests`, respectively. Then, leafnodes in `eproj/tests/eproj/net_tests` would be referenced with a python import path starting with `eproj.net_tests`, and those in `fproj/tests/fproj/net_tests` would be referenced starting with `fproj.net_tests`.

3.3.3 Running Leafnodes

This section deals with running leafnodes apart from the automated workflow and range management features provided by overmind. For range testing, see the sections on Test Plans and Remote Commit.

3.3.3.1 Single Tests

Undermine, provided in the tybase repository, is the primary tool used to run leafnodes. Undermine lets a user run a single instance of a test script against one or more specific resources. The basic usage of undermine is as follows:

```
bin/undermine [--presetup <script_spec>] [--postcleanup <script_spec>]
<leaf_spec> <host_spec> [<host_spec> ...] --
            <args_and_kwargs>
```

The command line components are as follows:

- `script_spec`: This is an optional specification of the pre-setup or post-cleanup leafnode to run before or after the main leafnode runs, respectively. It is a list of four-tuples. It is essentially the parts of an undermine command line necessary to run an undermine script. For example:

  ```
  "[('leaf.setup1', [‘hosts[0]’], [‘arg1’, ‘arg2’], {‘kwarg1’:
  ‘val’, ‘kwarg2’: ‘val2’}), ('leaf.setup2', [‘hosts[1]’], [],
  {})]"
  ```

The first argument in the tuple is a string of the leaf module. This must be an accessible module from undermine's python import path. In other words, from the example above, you should be able to run the same script in undermine:

```
bin/undermine leaf.setup1 10.11.12.13 10.11.12.14
```

The second argument is a list of strings for the host index values. The index values must be within range of the number of host_specs supplied to the main test script.

The third argument is a list of arguments as strings. The fourth argument is a dictionary of keyword arguments (kwags) with key/value pair values as strings.

An example undermine command line using pre-setup and post-cleanup specifications is as follows (the command has been newline separated and indented for readability):

```
bin/undermine
  –-presetup "[('leaf.setup1', [‘hosts[0]’], [‘arg1’, ‘arg2’],
                  {‘kwarg1’: ‘val’, ‘kwarg2’: ‘val2’}),
              ('leaf.setup2', [‘hosts[1]’], [], {})]"
  –-postcleanup "[('leaf.cleanup1', [‘hosts[0]’], [], {}),
              ('leaf.cleanup2', [‘hosts[1]’], [], {})]"
  some.script.test 10.11.12.13 10.11.12.14
```

- `leaf_spec`: This is the specification of the leafnode to run, and may be given as one of the following:
  - python import path to the leafnode: Since all leafbags are on the python path, you can give the "python import path" to your leafnode as if you were in python code trying to import it. For example, suppose you have a leafnode called ping in the file `leafbags/my_leafbag/utils/network_funcs.py` (relative to the root of tybase). Then, since `leafbags/my_leafbag` is on the python path, if you were in python code and wanted to import your ping leafnode, you would say `import utils.network_funcs.ping`. Therefore, to run this leafnode, you would use `utils.network_funcs.ping` as the `leaf_spec`. If you put the `MainLeaf` decorator on your ping leafnode, then you could get away with just `utils.network_funcs`.
  - filesystem path: An alternative way is to specify the filesystem path to the leafnode. This *may* allow you to run leafnodes even when they're not in leafbags (caveat emptor). To do this, you just give the filesystem path to the python file containing the leafnode, followed by the name of the leafnode in the file, with an '@' in between. For example: `leafbags/my_leafbag/utils/network_funcs.py@ping`. As with the python import path, if you put the `MainLeaf` decorator on the ping leafnode, you can leave off the `@ping` component.
- `host_spec`: This is a specification of the host to connect to with palantir. Since palantir currently only runs over TCP/IP, this must be either the IP address or hostname of the host.

- `args_and_kwargs`: Here you specify, space-delimited, the arguments and keyword arguments to the leafnode. See the output of `bin/undermine -h` for an extensive description of exactly how arguments and keyword arguments are specified.

When undermine runs it logs into two files: script.log and undermine.log. script.log contains logging and output specifically from the test script. undermine.log contains lower-level logging from undermine and palantir, logging which the test script developer may not care about. These files are located in the script's output directory.

When running test scripts standalone with undermine, by default, output directories are stored under `output/undermine/`<USERNAME> relative to the tybase root, where <USERNAME> is the user name of the user running undermine. Under this directory, a subdirectory named with the current date and time is created, and this timestamp directory is the output directory of the leafnode. Also, in the <USERNAME> directory will be a symlink called "latest" which always points to the most-recently-run leafnode. This is especially helpful when there are lots of output directories sitting around in a <USERNAME> directory.

It is important to note that running undermine runs a test on a resource in its current state. In other words, the resource is not automatically reverted to a clean state. If you wish to run an undermine test on a machine in a clean (or previous state) you must manually restore the state of the resource before running a test. For virtual machines, revert the machine to the desired snapshot. For physical machines, you must restore a valid image to the resource using `bin/comp_admin`'s `restore_state command`. For physical machines, this command runs clonezilla to restore an image to the machine. Currently, any clonezilla operations must be run on the main Tyrant server (not a machine where a tester would perform remote commit) To restore a resource to a clean or previous state, run the following command from tybase root:

```
bin/comp_admin name=test_comp restore_state snapshot_name
```

NOTE: if there are special characters or spaces in the computer name or snapshot name (i.e. "test_comp" is "test comp"), you must surround the name with escaped quotes:

```
bin/comp_admin name=\"test comp\" restore_state snapshot_name
```

Some tests on physical machine resources may require a wiped drive before running. If this is the case, you must set (or add) the configuration `pre_restore_wipe` to `True` in `rc/hal.rc` on the main Tyrant server.

3.3.3.2 Batch Testing

The plundermine tool provided in tybase allows running simple combinations of tests against specific test resources. To use it, you give plundermine a leafnode to run, and lists of hosts and parameters for each host and parameter slot the leafnode takes. Plundermine will generate all the possible combinations and run them with a level of parallelism (up to a configurable maximum number of concurrent undermine runs). In the fairly common degenerate case of a leafnode which accepts only

one host and no arguments, plundermine is an effective tool for running a given leafnode against a whole set of resources. This is useful for some range management tasks.

See the output of `bin/plundermine –h` for full details. Some examples are:

- Run a leafnode which only accepts one host against the three listed hosts:
  ```
  bin/plundermine underlib.test_leafnode
  192.168.56.1,192.168.56.2,192.168.56.3
  ```
- Run a leafnode which accepts two hosts and no arguments against all possible combinations of hosts listed in the two specified files:
  ```
  bin/plundermine underlib.client_server_test file:clients
      file:servers
  ```
- Run a leafnode which accepts two hosts and two arguments against all possible combinations of the hosts from the first file, the hosts from the comma-separated list for the second host slot, and the specified values for the two argument slots:
  ```
  bin/plundermine underlib.complex_test file:first_hosts
      192.168.56.1,192.168.56.2 -- one,two,three x,y,z
  ```

For plundermine, output directories are stored in `output/plundermine/<USERNAME-TIMESTAMP>`, where <USERNAME> is the name of the user who ran plundermine, and <TIMESTAMP> is the date and time at which plundermine was run. Inside each of these directories are numbered subdirectories representing each of the test instances run. These numbered subdirectories are each undermine output directories containing the undermine log and data files.

3.3.3.3 Scheduling Future Tests

In order to schedule regular tests navigate to the Test Scheduler (Tests ␣ Test Scheduler).

ATTENTION: Ensure that the Test Scheduler setup has been completed as described in the administrator manual.

The Test Scheduler is the Tyrant tool for kicking off test plans as a cron job. Some features of the Test Scheduler (notably the time entry select box) are motivated by cron's design.

To schedule a job:

- Enter the absolute path to the tyworkflow repository
  - Ex. /proj/testing/commits/user1/tyworkflow
- Enter the absolute path to the test plan to be scheduled
  - Ex. /proj/testing/commits/user1/tyworkflow/src/leafbag/overlib/preflight/service_ping_plan.py
- Use the select boxes to capture the regular time that the test plan should be run in regular cron format. An empty box will be interpreted as "all."
- Optionally enter any desired test plan parameters in the "Set Test Plan Parameters"

panel

- Click “Create Job”

To remove a job simply find the job under “Scheduled Jobs” and click the “Remove” button on the right side.

The Test Scheduler jobs reside in /etc/cron.d/ATL-Tyrant-Cron-Scheduler.

4 Test Plans

Test plans are the units of work performed by Overmind. They specify what test to run (the leafnode the user already has) and what to run it on ("all versions of Windows", "all languages of Windows XP SP2", etc). Overmind utilizes the reaper to revert assets to previously stored state (typically, reverting to a snapshot on a VM). Overmind stores the results of undermine runs in a database which can then be viewed through the overview web gui. Like leafnodes, test plans are written in python and stored in leafbags and are referenced on the command line in the same manner (either as python import paths or filesystem paths).

4.1.1 Test Plan Concepts

The following terms will be useful to know when writing and running test plans:

- test plan: A python script which defines test cases. This is the thing you run with overmind.
- test case: A description of what leafnode to run, on what assets, with what parameters.
- combo or test instance: A specific combination of leafnode, assets and parameters. A test case expands into multiple test instances at run time. These test instances represent individual runs of undermine. NOTE: In overview, the overmind web gui, test instances are referred to as test cases.
- namespace: An overall identifier in overmind under which multiple plans can run.
- purge: To immediately cancel a running namespace, plan, or test instance.
- reap: To revert an asset to a previously stored state.
- recipe: A definition of an OS which can be placed on a computer (e.g. the family, service pack, architecture, language, installed apps)
- computer: A computer on which a recipe can be installed (e.g. a VM or physical machine)
- resource: A specific computer with a specific recipe on it.

4.1.2 Example Test Plan

In your tyworkflow repository, look at `src/leafbag/overlib/preflight/service_ping_plan.py`. This is a test plan which runs the `service_ping_test` on all unique combinations of computer and recipe in the range, which for our purposes is equivalent to all the resources on the range. We’ll walk through this line-by-line:

```
from tyworkflow.support.planlang import *
```

This line imports the plan language objects used in writing the test plan

```
test = TESTCASE (
```

We begin the definition of a test case which will be used to generate all the actual tests run.

```
        script = 'overlib.preflight.service_ping_test',
```

This defines what leafnode to run. You must use the "python import path" method of specifying the leafnode to run; do not use a filesystem path.

```
        hostslots = [HOST() % FACTORS(computer_id=1, recipe_id=1)],
```

This defines how to generate the actual tests run, or "combos". This hostslots setting indicates the test only takes one host (since a list of only one element is provided), puts no constraints on the chosen host (because of the empty argument list to HOST), and indicates that the combination of computer_id and recipe_id for each chosen host must be unique. This is covered in-depth later.

```
        samples = -1,
```

This line specifies how many of the generated combos to actually choose. The special value -1 indicates that all combos should be used.

```
        namespace = 'preflight-ping-$t',
```

This specifies the name of the namespace to use if no namespace is .

The values defined in the TESTCASE declaration may be overridden on the command line.

```
EXECUTE (
        testcase = test,
)
```

This block sets the testcase defined above to be execute.

4.1.3 Parsing and Solving Test Plans

Prior to actually running a test plan, there are a couple operations which may be performed on it to verify that it will do what you expect. Because you configured your database by editing `rc/db.rc` in tyworkflow earlier, you can run these steps locally even though your test plans will be run remotely.

The `process_plan` command provided in tyworkflow provides the parse and solve subcommands. Parse will parse your test plan and give back to you overmind's understanding of what you're written. You can use it as a quick verification that you wrote what you intended. To parse the example plan from above, you would run

```
bin/process_plan parse overlib.preflight.service_ping_plan
```

If you wanted to parse the plan but then override the samples setting, you could run

```
bin/process_plan parse overlib.preflight.service_ping_plan samples=10
```

and you would see that change reflected in the output.

The solve subcommand will parse your plan and then show you what combos your plan would generate. To solve the example plan, you'd run:

```
bin/process_plan solve overlib.preflight.service_ping_plan
```

You could override samples like above, and depending upon how diverse your range is, you may see the number of combos decrease. Note that combo generation involves randomization and depends upon the state of the range. For example, if you have a test plan that could generate a total of 100 combos, but you set samples to 20, then each time you run solve, you will see a random set of 20 out of the 100

total possible combos. Also, if certain machines become totally unavailable (i.e. reserved or marked fubar), then some combos that would have been generated had they been available will no longer be generated. This means that each time you run solve, you may get different results.

4.1.4 Running Test Plans

The remote commit command (`bin/remote_commit`) in tyworkflow allows you to submit your tests to a remote test server. Once your test is submitted, you can continue your development in your local environment without affecting the test you just submitted. You can even submit tests in parallel, allowing you to try one approach to solving a problem, submit a test of it, then try a different approach and submit a test for that approach in a different namespace. Here we explain how to run test plans, but remote commit has more functionality which is covered in detail later on.

Test plans are run though remote commit with either the `run` or `runlite` subcommands. These commands sync your local environment up to the test server and submit your test to your remote overmind instance. These commands take the same arguments as `process_plan`'s `solve` and `parse`. To submit the `service_ping_plan` with `remote_commit`, you'd run:

```
bin/remote_commit run overlib.preflight.service_ping
```

If you wanted to limit the number of samples, you would run

```
bin/remote_commit run overlib.preflight.service_ping samples=10
```

Another useful option is to specify some notes on the namespace with the n_notes argument:

```
bin/remote_commit run overlib.preflight.service_ping samples=10
n_notes="please work"
```

4.1.5 Working with a Range

As a developer, overview will be your primary interface to range testing. Overview lets you browse test results and reserve machines for use outside of normal automated testing.

4.1.5.1 Seeing Test Results by Namespace

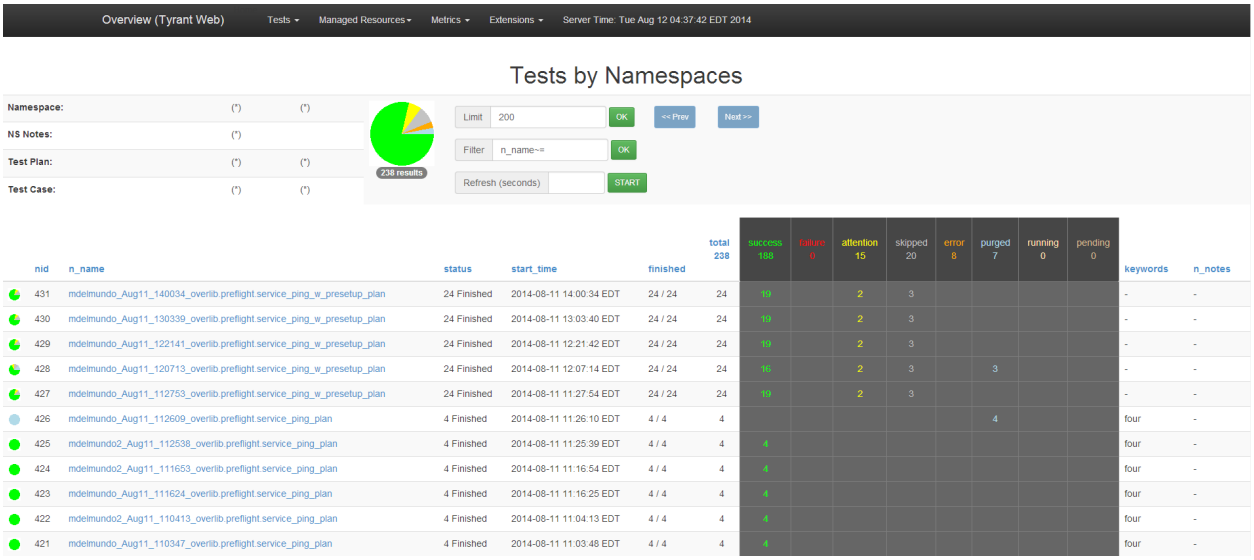
To see test results, navigate to overview's Namespaces page (e.g. http://testserver.example.com/test_namespaces.php). Here you'll see a listing of all the namespaces that have been run. When remote commit is in use, there will usually be only one plan in a namespace.

At each of the list levels, on the right side of the table, you'll see a summary of how many testcases in that entity are in the various states a test can be in (either pending, running, or one of the completion statuses).

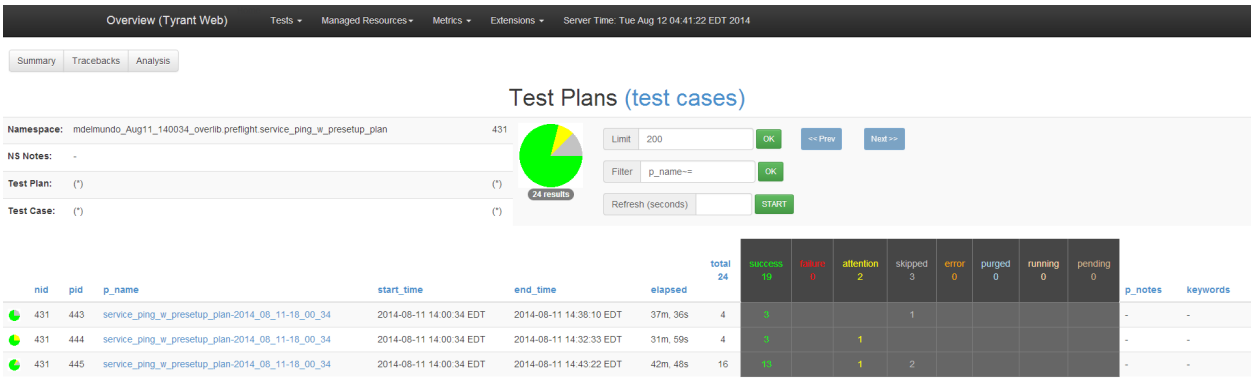
The list pages also have forms at the top allowing you to filter by matching on various attributes of namespaces, plans, or test cases (specified by naming the field of the table to filter on and the pattern to match, e.g. `status=error` to see all error testcases, `n_name~=preflight` to see all namespaces whose name matches the pattern "preflight", or keywords~=test1 to see all namespaces whose testplans include the keyword "test1").

The `Refresh` field specifies the interval in which any particular page will refresh. This is particularly useful for monitoring test results as they finish.

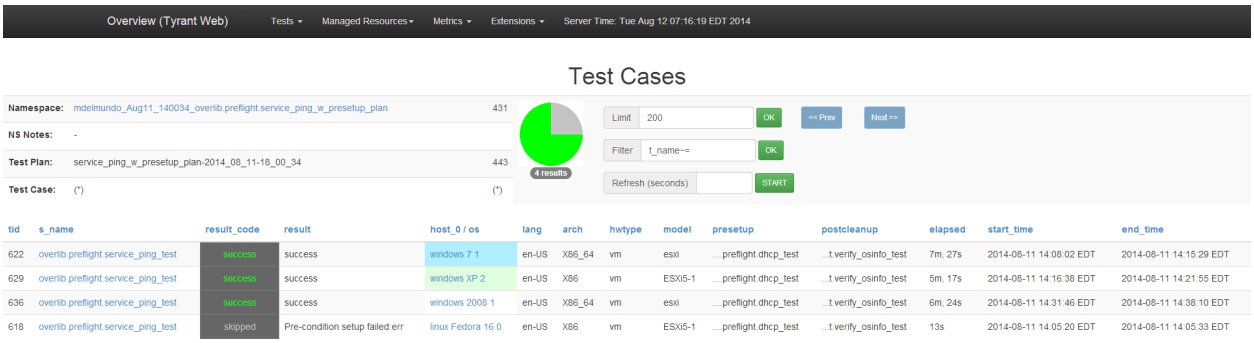
Finally, the `Limit` field allows you to specify how many rows you want to be displayed (whether namespaces, test plans, or test cases). This prevents loading an entire page with 1000's of rows if the loading time would take too long.



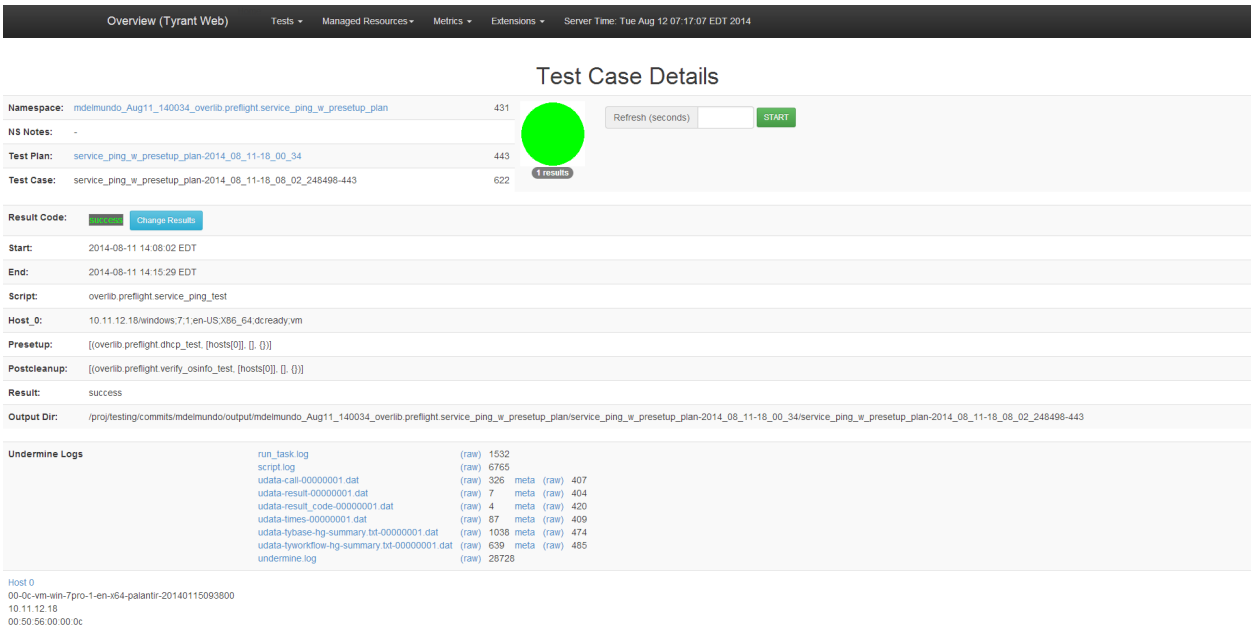
Clicking on a namespace brings you a list of all of the test plans in that namespace.



Clicking on a plan name brings you to a list of all the test cases in the plan.

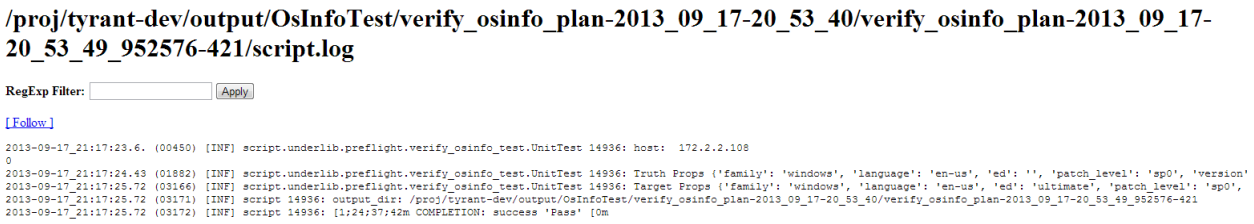


Clicking on the test case name gives you detailed results from that test.



The file `script.log` contains logging output specifically written by the test script developer (by calling `self.log` or `host.log` in a test script) which `undermine.log` gives lower-level undermine framework logging information. See these files in the case of errors or unexpected results with your tests scripts.

Clicking on any of the log files leads to the raw data from the file system.

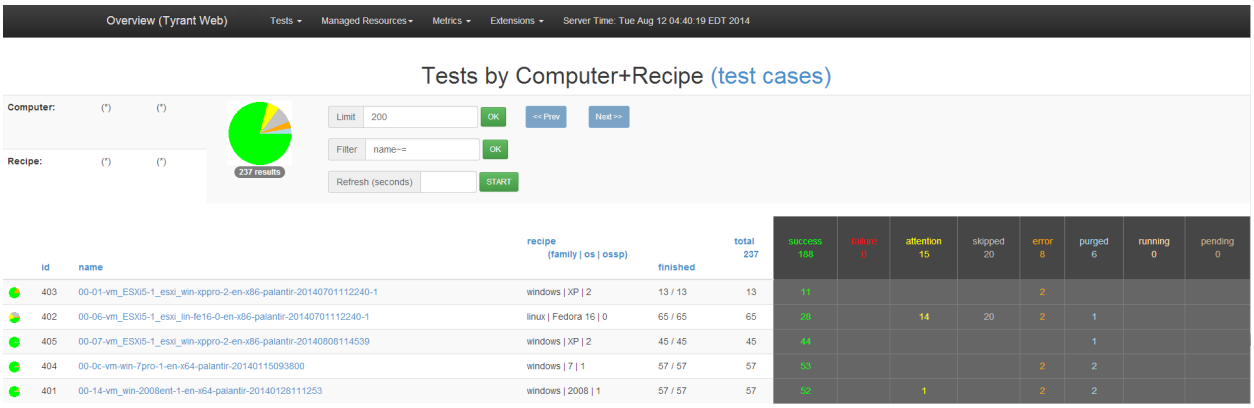


4.1.5.2 *Seeing Test Results by Computer+Recipe*

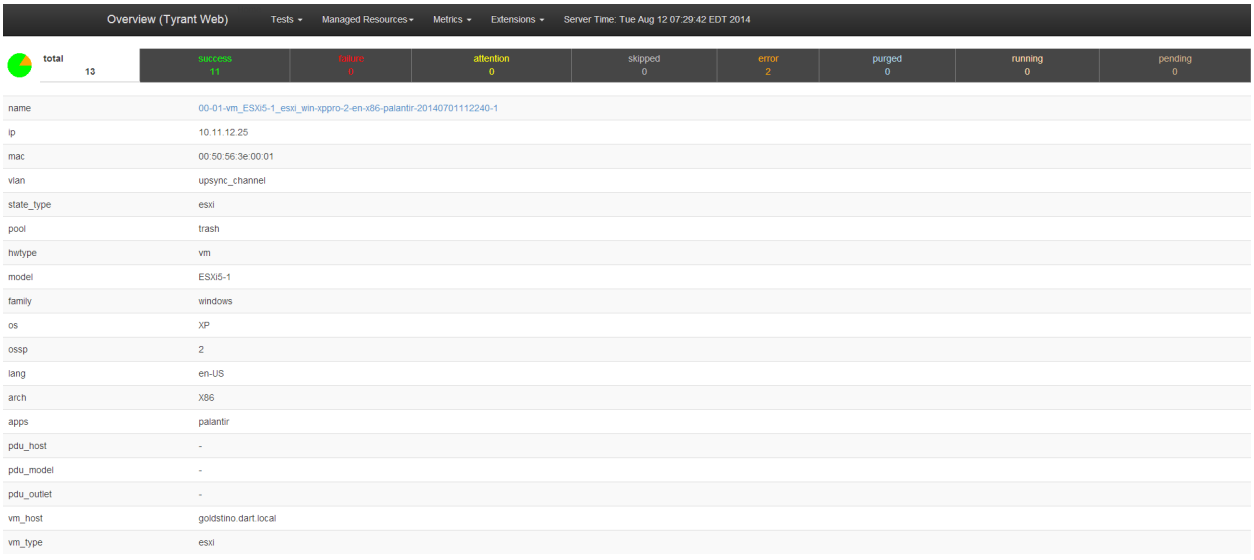
To see test results grouped by resource, navigate to overview's Tests by Resource page (e.g. http://testserver.example.com/test_resources.php). Here you'll see a listing of all the resources from test cases that have been run.

At each of the list levels, on the right side of the table, you'll see a summary of how many testcases in for that resources are in the various states a test can be in (either pending, running, or one of the completion statuses).

The list pages also have forms at the top allowing you to filter by matching on various attributes of a resource (specified by naming the field of the table to filter on and the pattern to match, e.g. `family=windows` to see all "windows" resources or `apps~=McAfee` to see all resources whose apps value matches the pattern "McAfee").



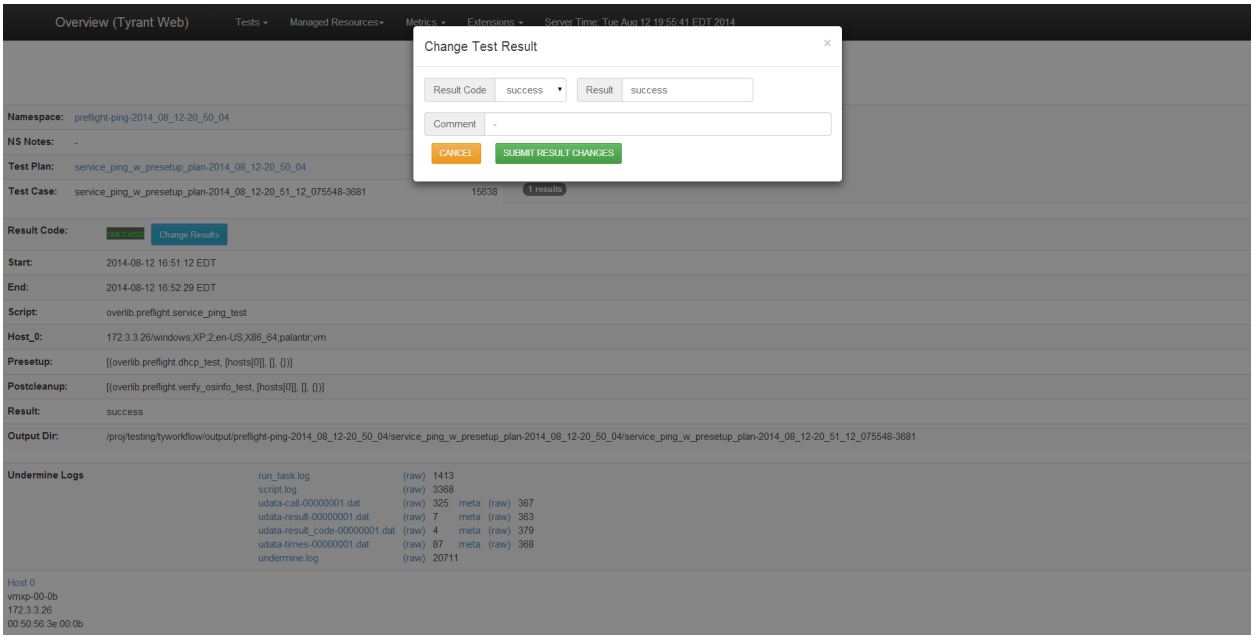
The test case computer details page shows a single row summary of the number of test case results for a resource. Clicking the pie chart in the summary table navigates to the test case computer page listing all test cases associated with that resource.



4.1.5.3 Changing Test Results

Tyrant users can change test results from the test case details page in Overview. Once a user submits a test result change, the original results and timestamp of the change are stored and the current result is marked as changed.

To change the result of a test case:



1. Navigate to the test case details page.
2. Click Change button next to the test result.
3. Select new test result and enter comment.
4. Click OK.

Once a result changes, the original result and timestamp of change appears on the test case details page. Additionally, the test case page displays an asterisk next to the result code if the result had been changed at any point. **NOTE**: Even if the test case result changes to a different result code and then changes back to its original result code, the test case result will still be marked as changed.

tid	s_name	result_code	result
651	overlib.preflight.service_ping_test	success	success
652	overlib.preflight.service_ping_test	success	success
653	overlib.preflight.service_ping_test	success	success
650	overlib.preflight.service_ping_test	*attention	changed

4.1.6 Test Plans in Depth

Within the previously outlined basic structure of plan files, there are many constructs in the "planlang" (provided in your plans by the line

```
from tyworkflow.support.planlang import *
```

present at the top of each plan file). Together, these constructs are used to build the specification of what tests to run, with what arguments and what types of test resources.

4.1.6.1 FILTER

This is a base class which supports abstract filtering based on values of named attributes. It is extended by some other classes in the planlang and is generally not used directly. The typical usage is the HOST subclass.

4.1.6.2 HOST

A `FILTER` subclass that filters resources based on attribute value constraints. The value can be a singleton or a list of values. The constraints are specified in the constructor with the general form:

```
HOST(<attr>=<value>|[<value>, <value>,...], <attr>=...)
```

For example,

```
HOST(family='windows', os=['2k', 'xp'], ossp='sp0')
```

`HOST` objects are the primary means to define the desired set of resources to use for a given test script. `HOST` objects are used in the `hostslots` argument to the `TESTCASE` constructor, explained later.

IMPORTANT: The values for "family" and "os" and all of the other fields on which a user can write a filter are set by the Overmind database. These values can be viewed using Overview's "Recipes" and "Resources" pages to determine what valid values are. For example, if the administrator sets the "os" of a box to be the string "xp_pro" instead of "xppro" and the user wants to run on XP boxes, the filter for "os" needs to be "xp_pro" – the exact string match. This loosely-defined schema is nice for rapidly adapting to new recipes; however, it does require coordination between the users and the administrators.

We recommend the schema style specified in the `recipes.csv` file in the `docs/` directory of tyworkflow; however, it is more important that an organization is consistent with whatever schema they choose.

4.1.6.3 FACTORS

The class used to control the sampling of test instances. For example, consider a plan with 3 host slots and a resource pool of 100 machines. In the worst case, this could generate 100^3 potential test instances. If each test takes 10 minutes, even with max parallelism of 33 simultaneous test instances, it would take a minimum of 10,000,000/33 minutes or ~21 days. In this case, you may want to sample the set. `FACTORS` objects specify the attributes of interest to vary across test instances. Other attributes will be randomly selected based on resource availability. The constructor defines the attributes of interest with the general form:

```
FACTOR(<attr>=True|False, ...)
```

For example,

```
FACTOR(family=True, os=True, ossp=True, lang=True)
```

Note that you must set all keyword arguments to FACTOR to either `True` or `False`. In other words, you are limited to either specifying the set of fields you care about or the set of fields you don't care about.

4.1.6.4 TESTCASE

The class used to compose HOST, FACTOR, and parameter values to form a specification for a set of test instances. Specifically, a TESTCASE constructor takes:

- `script`: Name of test script (leafnode).
- `hostslots`: List of FILTER objects, defining the number of host resources and their constraints. For example:

  ```
  [HOST(), HOST()]
  ```

  The number of elements in the `hostslots` list defines the number of resources each testcase will use and should match the number of resources the test script defined in the script argument requires.
- `paramslots`: List of parameter values (defined as a list). For example:

  ```
  [['a', 'b'], ['c']]
  ```

  Combos are generated for each potential value of a parameter slot. In the given example, the first parameter slot can be either 'a' or 'b', but the second parameter slot will always be 'c'. So, for a very simple example range with only one test resource, a plan with this `paramslots` setting would generate two combos, one with arguments 'a' and 'c', the other with arguments 'b' and 'c'.
- `filter`: Singleton FILTER object that defines a global constraint over resources. For example:

  ```
  HOST(pool='pname')
  ```
- `xattrs`: Singleton XATTRS object, defining attribute constraints across host objects. For example:

  ```
  XATTRS(vlan='same')
  ```

  This line would ensure that the hosts are on the same subnet.
- `factors`: Singleton FACTORS object, defining sampling attributes. For example:

  ```
  FACTORS(os=1)
  ```
- `samples`: Maximum number of sample test instances to run.
- `replications`: Number of times to run each sample.
- `priority`: Numeric priority of test instances (used to sort scheduling queue).
- `namespace`: Namespace name to use when storing test results in database.
- `post_ops`: List of functions to run as post-test operations. As previously mentioned, these post-ops will be run sequentially for each resource used in a test, in the context of the Overmind server, after the Undermine process executing your test script exits. In this way, they differ from the `presetup`/`postcleanup` options further down in this list. Each element of this list is a tuple consisting of a reference to the function to be run, and a list of arguments to be given to the function. Each post-op receives a reference to the task object representing an instance of a test, and whatever arguments are specified in the test plan (i.e. the arguments in the tuple).

Examples of post-ops can be found in tyworkflow at `src/tyworkflow/support/planlang.py`, starting with the RESERVE_ON function.

- `n_notes`: Informational notes to store with the namespace this testcase will run in. Remember, this MUST be quoted if it contains spaces.
- `p_notes`: Informational notes to store with the test plan this testcase will run in. Remember, this MUST be quoted if it contains spaces.
- `keywords`: A comma separated list of strings. Test plans can be searched and filtered based on their keywords.
- `presetup`: List of pre-setup script values (defined as a list of four-tuples). For example:

```
[
  [('leaf.setup1',['hosts[0]'],['arg1', 'arg2'],
      {'kwarg1': 'val1', 'kwarg2': 'val2'}),
   ('leaf.setup2',['hosts[1]'],[],{})
  ],
  [('leaf.setup2',['hosts[1]'],[],{})
  ]
]
```

Combos are generated for each potential value of a pre-setup script. In the given example, a test would run the leaf.setup1 AND leaf.setup2 script OR only run the leaf.setup2 script before its main test script.

- `postcleanup`: List of post-cleanup script values (defined as a list of four-tuples). For example:

```
[
  [('leaf.cleanup1',['hosts[0]'],['arg1', 'arg2'],
      {'kwarg1': 'val1', 'kwarg2': 'val2'}),
   ('leaf.cleanup2',['hosts[1]'],[],{})
  ],
  [('leaf.cleanup2',['hosts[1]'],[],{})
  ]
]
```

Combos are generated for each potential value of a post-cleanup script. In the given example, a test would run the leaf.cleanup1 AND leaf.cleanup2 script OR only run the leaf.cleanup2 script after its main test script.

4.1.6.4.1 Cloning Resources Based on Test Results

One specific example of the `post_ops` TESTCASE parameter is the post-op which clones the resources used in a test if the test exits with a certain result code. For example, you may want to clone the resources when a test returns `FAILURE`. To use this feature, you would add the following to your testplan, inside the TESTCASE constructor:

```
post_ops = [(CLONE_ON, 'FAILURE')],
```

For an example in context, see `src/leafbag/overlib/preflight/clone_plan.py` in tyworkflow.

4.1.6.5 EXECUTE

The class used to define which TESTCASEs to run for the plan file. The separation of TESTCASE and EXECUTE allows you to compose TESTCASE separately from specifying which ones to run. The EXECUTE constructor takes the TESTCASE object and an optional set of keyword arguments. If the optional TESTCASE keyword arguments are provided, they are used to override the value for the given TESTCASE. Note this is purely for convenience and `EXECUTE(testcase, **plan_ops)` is equivalent to:

```
EXECUTE(testcase / TESTCASE(**plan_ops))
```

4.1.6.6 PARSE

Returns the list of EXECUTE'd TESTCASEs for a given plan file. Any list operator can be applied to this list for composing plans from plans. As with the EXECUTE statement optional arguments can be provided to override the value for the resulting TESTCASEs. Note this is purely for convenience and `PARSE(plan_file, **plan_ops)` is equivalent to:

```
map(lambda x: x/TESTCASE(**plan_ops), PARSE(plan_file))
```

4.1.6.7 Planlang Operators

The language provides a set of operators over objects for plan reuse and simplification. For `FILTER` operators it is best to think of a `FILTER` object as representing a set, specifically, the set of resources that satisfy the constraints. The valid operators are:

- `FILTER & FILTER`: A new `FILTER` object representing the set intersection of the two.
- `FILTER | FILTER`: A new `FILTER` object representing the set union of the two.
- `FILTER - FILTER`: A new `FILTER` object representing the set subtraction of the two.
- `FILTER % FACTORS`: Binds the `FACTORS` to the `FILTER` object
- `FILTER % XATTRS`: Binds the XATTRS to the `FILTER` object
- `TESTCASE / TESTCASE`: Copy of left TESTCASE with attributes overridden with the defined attributes of the right TESTCASE (undefined attributes use the value from the left TESTCASE).
- `TESTCASE & FILTER`: Copy of TESTCASE with global filter constraint intersected with `FILTER`.
- `TESTCASE | FILTER`: Copy of TESTCASE with global filter constraint unioned with `FILTER`.
- `TESTCASE - FILTER`: Copy of TESTCASE with global filter subtracted with `FILTER`.
- `FACTORS + FACTORS`: A `FACTORS` instance with the union of attributes from both operands.
- `XATTRS + XATTRS`: An XATTRS instance with the union of attributes from both operands, with attributes in the second operand overriding those in the first when the same attribute is present in both operands.

4.1.7 Plans of Plans

One incredibly useful feature is to generate plans of plans. A typical use case is to have a high level plan that kicks off a lot more lower level plans. For example, a regression_test plan could include kicking off all of the relevant plans for a single situation.

To do this, one needs to generate a plan file that calls `run_plan` on other plans. The arguments to run_plan are the following:

- `namespace`: typically `globals()`
- `search_path`: typically the leaf bag containing the plans
- `plan_name`: specific plan name to be called by this plan (without ".py" extension)
- `maxCount`: how many times to run this plan (equivalent to setting "samples=" from the remote_commit command line

For example,

```
from tyworkflow.support.planlakng import run_plan

run_plan(globals(), "my.leafbag", "MyPlan", 5)
run_plan(globals(), "my.leafbag", "MyPlan2", 2)
```

NOTE: Any command line arguments (e.g. "samples=") passed to remote_commit will override any arguments passed to the test plans themselves via `run_plan`.

4.1.8 Remote Commit in Depth

Remote commit is provided as an alternative method of setting up an overmind test range and submitting plans to it which is useful for environments of multiple developers working against a single overmind-controlled test range.

For background, the standard (non-remote-commit) way of doing things is to, on a single machine, run the overmind, reaper, and overview servers. The user may choose to set up a MySQL database for overmind to store information about assets and test results in, or they may use the default SQLite database. The user then links in their testing code as a leafbag in their tybase repository and submits plans from their tyworkflow repository (which has tybase linked in at media/tybase). This lends itself well to a single developer working on the range, but doesn't work so well for multiple developers on different workstations. Under this model, they would have to SSH in to the machine running overmind, cd to the appropriate tyrant directory, put their testing code in place, and then submit a plan. Parallel work by multiple developers is not favored in this model.

In contrast, remote commit gets around these problems. With remote commit, a single MySQL database serves as the point of concurrency. Global instances of reaper and overview are run on a server and a directory is created on this server to serve as the root for remote committing to. Users set up their own local working directories of tybase, tyworkflow, and their SUT(s). When the user submits a test plan with remote commit, remote commit rsyncs all their testing code up to the server (in a subdirectory for the developer) and runs the overmind in the tyworkflow the user synced up to schedule the user's tests

(referring to the database to see what's available). The results of the user's tests are recorded to the global MySQL database, which all the users can view on the global overview instance. Any files that resulted from the test are also stored on disk and accessible from the overview interface. Once the user has run remote commit, which only takes as long as is needed to sync their code up to the server, they user may continue development in their working directory without affecting the tests that are running on the server. By specifying alternate usernames when submitting test plans, the user may even submit one test plan while another is already running.

4.1.8.1 SUT Preparation

In order to use a given SUT with remote commit, you must prepare a shell script that provides some functions and constants which remote commit will use. These should be placed in a file named rcoverrides.sh in the root of your SUT's leafbag. It is important that the functions respect posix conventions for return code, i.e. return 0 on success, nonzero on error. All override functions run relative to the tyworkflow root. Likely, you'll want some of your functions to run relative to the tybase root. In that case, you'll need to do something like `pushd media/tybase` at the top of your function and then `popd` at the end.

Functions:

- `_rcoverride_build`: Performs any steps necessary to build the SUT or prepare it for testing that are not encompassed in the actual testing code. For example, if your SUT is a single C file that requires compilation before testing, your `_rcoverride_build` function might just run gcc. If your project is more complicated and has a makefile, your `_rcoverride_build` could run `make`.
- `_rcoverride_clobber`: Called immediately after running a `make clobber` in tyworkflow's root, as part of the `clobber` command. Performs a thorough cleanup of the SUT directory. Continuing on the example from the previous function, the `_rcoverride_clobber` for a simple one-file C SUT might just delete the binary resulting from the compilation encoded in `_rcoverride_build`, while the implementation for a project with a makefile might run `make clean`.
- `_rcoverride_stop`: Called when running the stop command, immediately after shutting down the overmind service.
- `_rcoverride_summary`: Called when running the summary command, just after running `bin/scan_output` (which summarizes the undermine runs recorded in the output directory. This function allows you to add any custom output to the summary output.
- `_rcoverride_submit`: Called during running of the submit command on the testing server, immediately prior to actually submitting the desired plan to the running overmind. This allows you to do things like make settings changes to the running overmind conditionally based on which test is being run (a specific example would be to increase the maximum number of children for certain larger test plans).

Constants:

- `UNDERMINES`: Overrides the default maximum number of children your remote overmind process will allow.
- `CLIENT_TIMEOUT`: Overrides the default client timeout (maximum running time for various interactions with overmind).

4.1.8.1.1 Leafbags with non-leafbag dependencies

In order for tests to work with remote commit, the tests and all their dependencies must be linked in to your environment so that they will all end up on the testing server during remote commit's rsync. For the case of a self-contained leafbag, this works trivially: your SUT's leafbag is linked in to your tybase repository (in `leafbags/`), so when remote commit syncs, the leafbag is synced up. When your leafbag has non-leafbag dependencies, those dependencies must also be linked in to your copy of tyrant so that they will also be synced up during remote commit, and your testing code must be written so that it can reference the dependency relative to the tybase root (so that there are no hardcoded paths that will break when the entire directory is synced to some other system). Link in your extra dependency with a symlink in tybase's `media` directory, then write your testing code to refer to the supporting components in that location. Then, using the `search_media_path` function provided by in `tybase.support.util`, you can retrieve the path to the directory you linked in and then do whatever you need to with it (e.g. open a file relative to it, add it to the python path so you can import from it, etc). Call `search_media_path` with the name of the symlink you created inside of media, and it will return a usable path to your linked-in media.

4.1.8.1.1.1 Example

Suppose you have a project with a repository called eproj which contains the following two subdirectories (among others): `leafbag` (the leafbag with leafnodes for your project) and `data` (a directory with some sort of supporting files that are used both for your tests scripts in leafbag and by other parts of the software). You would link the leafbag in to tyrant's `leafbags/` as always. Then, you would also put a symlink in tyrant's `media/` pointing to eproj's `data` subdirectory. For example, assuming your current working directory is the root of your tyrant copy and eproj is a sibling of your tyrant copy, you could run

```
ln -s ../../eproj/data media/eproj-data
```

Then, a hypothetical eproj test script might contain code like the following to make use of files in that directory:

```
from tybase.support.util import search_media_path
epdata = search_media_path('eproj-data')
with open(os.path.join(epdata, 'seed-001.txt'), 'rb') as seedfh:
     seed_dat = seedfh.read()
```

4.1.8.1.2 Shared directories

It may be that your testing involves some large set of files which don't change very much and can be shared among developers. While each developer does need their own copy of these files on their local workstation for any local tests they might be doing, you would rather not have multiple copies of this large set of files on the testing server (since each developer has their own subdirectory of the commits

directory to which they would have to sync their own separate copy of the shared files) and would rather not have to go through syncing those files every time a developer runs a test.

Remote commit provides a way to handle this. In the `remote_commit.rc` file, create a section called `shared_dirs`. Each option in this section defines a single shared directory. The name of the option is the path to the shared directory relative to the root of tybase (i.e. where the shared directory currently resides in the tybase clone in your local testing environment). The value of the option is the path to the actual shared directory on the testing server. When you do a remote commit operation that involves a sync (`run`, `runlite`, `sync`, or `synclite`), the paths given as the option names in the `shared_dirs` section are excluded from the sync. After the rsync part of the sync process is complete, remote commit will create symlinks inside the remote tybase root (at the paths given as the option names) pointing to the paths given as the values to those option names. This saves having multiple copies of the large shared files on the testing server, however it is your responsiblity to make sure the copy of the shared directory on the testing server is kept up to date, since that will not happen automatically when a developer does a “sync”.

4.1.8.1.2.1 Example

Suppose your eproj test scripts require a set of installers and data files that exist in a directory called `eproj_data`. Currently, you make this set of files accessible by putting a symlink named `eproj_data` in tybase's media directory which points to the actual `eproj_data` directory (so the location of that symlink relative to the root of tybase is `media/eproj_data`) and then writing your test scripts to access the files out of media. To set this up as a shared directory for remote commit, you would do the following:

- Place a copy of the shared directory somewhere on the testing server. For our example, we'll put it at `/proj/eproj_data`.
- Add an option to the `shared_dirs` section of tyworkflow’s `rc/remote_commit.rc` whose name is the location of the shared directory inside tybase and whose value is the location of the shared directory on the testing server. For our example, this is:

  ```
  media/eproj_data = /proj/eproj_data
  ```
- Now, when you do a sync, `media/eproj_data` will be excluded from the initial rsync operation. Then, after that operation completes, a symlink will be created inside the remote tybase at `media/eproj_data` pointing to `/proj/eproj_data`.

4.1.8.2 Usage

Remote commit provides several commands via the `bin/remote_commit` script (in tyworkflow) which are used to sync your SUT up to the testing server, run tests, and administer your remote instance of overmind. These commands are explained below grouped by use case

4.1.8.2.1 (Dry)Running Tests (run, runlite, submit, parse, solve)

The `run` and `runlite` commands are the backbone of running remote commit; you can ignore all the others and still work effectively with these two commands. They are basically wrappers which run several commands under the hood. They both sync your testing environment up to the testing server,

start your remote instance of overmind, and then submit the given plan. The difference between the lite and full versions is that the full version does a clobber and build on the local side before syncing up to the testing server, whereas the lite version does not do this.

Usage:

```
bin/remote_commit [-u <USER>] run <PLAN> [<process_plan_opts>]
bin/remote_commit [-u <USER>] runlite <PLAN>
[<process_plan_opts>]
```

where <PLAN> is the designator for a plan file (either a python import path or filesystem path). The `-u` option specifies the subdirectory of the commits directory to work out of, and defaults to the current username (of the person running `bin/remote_commit`). In the case of run, the developer's testing environment will be synced up to the named subdirectory of the commits directory, the overmind in that directory will be used, etc.

Also, since `run` and `runlite` end up calling `bin/process_plan` on the remote side, you may override certain testcase attributes just as you would if you were calling `bin/process_plan` directly. This ability to override testcase attributes is available for any of the remote_commit commands that call `bin/process_plan` on the remote side. The one we find most useful is to override the samples value to run a subset of a potentially large set of combos. For example, your testplan may generate 100 combos, but you only want to run a random ten of them. Then, you would do:

```
bin/remote_commit run PLAN samples=10
```

The `submit` command simply syncs up your testing environment and submits the given plan without running the build step. This is useful if you make local changes ONLY to your SUT's testing code (or other minor local changes which don't require rebuilding your SUT), because in that case the changes you're syncing up won't have any effect on your remote overmind instance, so a restart is not necessary. If, however, you've made SUT changes that require a rebuild, then `submit` may not be safe to run; you should use `run` and `runlite` instead. If you've made changes to actual tyrant code, then you need to do the `sync` command (explained later) first to force your remote overmind instance to restart.

Usage:

```
bin/remote_commit [-u <USER>] submit <PLAN> [<process_plan_opts>]
```

The `solve` command allows you to see how many combos your test plan will run when submitted. This is analogous to the `bin/process_plan solve` command used with the classical overmind setup.

Usage:

```
bin/remote_commit [-u <USER>] solve <PLAN> [<process_plan_opts>]
```

4.1.8.2.2 Syncing your testing environment (sync, synclite, diff)

These two commands sync your testing environment up to the testing server. As with run and runlite, the full version does a local clobber and build, the lite version does not. An added difference between `sync` and `synclite` is that `sync` forces your remote overmind instance to restart, whereas

`synclite` does not. If you make changes to core tyrant code and want that to take effect on the remote side, you need to use `sync`, since if the remote overmind doesn't restart, your changes may not take effect, depending upon what you changed. Generally, developers and testers won't be making changes to overmind, but if you receive an updated delivery of tyrant code or the administrator makes some changes, you may need to run a full `sync`.

Usage:

```
bin/remote_commit [-u <USER>] sync
bin/remote_commit [-u <USER>] synclite
```

The `diff` command is provided as a dry run of syncing. It just uses rsync's dry run capability to show you what files will be uploaded/changed/deleted when syncing to the testing server.

Usage:

```
bin/remote_commit [-u <USER>] diff
```

4.1.8.2.3 Administering your remote overmind instance (start, stop, restart, set_children, get_children, set_popthreads, get_popthreads)

To start, stop, or restart your remote overmind instance, the respective commands are provided. Users do not typically use these since they are handled automatically when running `run` or `sync` commands. During the stop command, the custom `_rcoverride_stop` function is run, if provided.

Usage:

```
bin/remote_commit [-u <USER>] start
bin/remote_commit [-u <USER>] stop
bin/remote_commit [-u <USER>] restart
```

The `set_children` command allows you to set the maximum number of undermine processes your remote overmind process will run in parallel.

Usage:

```
bin/remote_commit [-u <USER>] set_children <NUM_CHILDREN>
```

where `NUM_CHILDREN` is an integer telling how many children to run in parallel.

The `get_children` command tells you the current max number of undermine processes.

Usage:

```
bin/remote_commit [-u <USER>] get_children
```

The `set_popthreads` and `get_popthreads`, let you, respectively, set and get the maximum number of concurrent post-op threads which will run.

Usage:

```
bin/remote_commit [-u <USER>] set_popthreads <NUM_POPTHREADS>
bin/remote_commit [-u <USER>] get_popthreads
```

4.1.8.2.4 Building and clobbering your SUT (build, clobber, rclobber)

The `build` and `clobber` commands run the `_rcoverride_build` and `_rcoverride_clobber` functions you provide in your `rcoverrides.sh` file; in other words, they locally build and clobber your SUT. The `rclobber` command runs the `_rcoverride_clobber` function, but on the remote testing environment.

Usage:

```
bin/remote_commit [-u <USER>] build
bin/remote_commit [-u <USER>] clobber
bin/remote_commit [-u <USER>] rclobber
```

4.1.8.2.5 Seeing results (summary)

In addition to viewing results of tests via the overview GUI, you can also use remote commit's summary command, which prints out a summary of your remote testing environment's output directory. If provided, the `_rcoverride_summary` function is also run after printing the default summary information.

Usage:

```
bin/remote_commit [-u <USER>] summary
```

4.1.8.2.6 Other commands (client)

`-u <USER>`The `client` command runs an arbitrary command with the overmind client (`bin/overmind_admin`) on the remote overmind instance.

Usage:

```
bin/remote_commit [-u <USER>] client <CMD>
```

where `<CMD>` is the command you want to run. See `bin/overmind_admin -h` for a list of potential commands to run.

4.1.9 Automatically Generating Plans

Sometimes, it's useful to run a test script with certain types of assets and certain parameters without having to write a plan file. To support this, the autoplan tool is provided. This tool allows you to automatically generate test plans based on command line parameters. To use autoplan via remote commit, the subcommands autoparse, autosolve, autosubmit, autorun, and autorunlite are exposed. These commands are analogous to the normal parse, solve, submit, run, and runlite commands, except that they use an autogenerated plan instead of a pre-written one.

To use autoplan, you specify the name of a test script to run (either filesystem path or python import notation), filters to describe what kind of asset is needed for each hostslot the script accepts, and potential values for each parameter slot the script accepts. Optionally, plan processing arguments (such as samples, n_notes, etc) may be specified as keyword arguments.

For example suppose you have a test script called "mywidget.command_test". Suppose this test script takes three assets: one running Windows 7 Ultimate SP2 64-bit, the other running Windows XP Professional SP3 32-bit, and the third being any asset. Suppose also that this test script accepts two

arguments, and you wish to run test plans where the first argument is either “yes” or “no” and the second is a number from 1 to 5. Finally, suppose you wish to run two replications of each test case. To automatically generate a test plan according to these specifications and then see it parsed to ensure it does what you want, you could run the following command:

```
bin/remote_commit autoparse mywidget.command_test -H
“os=’7ult’,ossp=sp2,arch=x64” -H os=xppro,ossp=sp3 -H ‘’
-p yes,no -p 1,2,3,4,5 replications=2
```

If you wanted to actually see what resources it would use, you could run the same command, but with the `autosolve` subcommand instead of `autoparse`. If you wanted to submit this autogenerated plan, without purging any currently running tests in your remote commit namespace, you could run the same command, but with `autosubmit` instead of `autoparse`.

To submit the autogenerated plan in the normal method (where any currently running tests in your remote commit userspace are purged and your new plan is submitted), use the autorun or autorunlite subcommands instead of autoparse. Like the normal run and runlite, run will perform a clobber and build on the local side before syncing to the testing server, whereas runlite will not.

Note also that you can parse and solve automatically generated plans locally without having to contact the testing server. To do this, instead of running…

```
bin/remote_commit autoparse …
```

…run this command…

```
bin/autoplan parse …
```

(and to do a local solve, use “solve” instead of “parse”).

Autoplan does support a few other lesser-used subcommands than those listed here. See `bin/autoplan -h` (run from tyworkflow) for more information.

4.1.9.1 Quoting with Autoplan

Due to the interaction of the shell and the tyrant commandline parser, certain strings in hostslots or parameter slots unfortunately must be quoted in special ways. If a host slot field value or a parameter slot value contain any characters other than letters, numbers, or underscores, or if one of these values begins with numbers, those values must be quoted. Furthermore, since the shell normally strips quotes, the overall hostslot or parameter slot argument must be double-quoted, or a single set of quotes must be escaped.

For example, suppose you have a test script which requires XP Professional assets. You would specify a host slot as follows:

```
-H os=xppro
```

Nothing too unusual here. If, however, you want to run a test script with 7 Professional assets (i.e. assets whose “os” field equals “7pro”), you must specify the host slot argument in one of the following ways:

```
-H os=\’7pro\’
-H os=\”7pro\”
-H “os=’7pro’”
-H ‘os=”7pro”’
```

In the first two cases, the sets of quotes are escaped so that the shell will not strip them. In the second two cases, the string is double-quoted. The shell will strip the outer set of quotes, but the inner set will make it into the tyrant commandline parser intact.

These rules hold for other scenarios, such as:

- a computer name with spaces and commas:
  ```
  -H “computer=’complicated, computer name’”
  ```
- a pool name with a dash
  ```
  -H pool=\’dash-pool\’
  ```
- a parameter slot value with spaces and dashes:
  ```
  -p “simpleval,’com-plex val’”
  ```

5 Appendix A - Event Detection

Via the tyutils add-on repository, Tyrant provides the ability to run arbitrary test scripts in a wrapper which monitors a test resource's screen for changes.

Tyutils provides two methods of acquiring screenshots: using native Windows functionality to take screenshots (which only works for Windows resources and can be affected by conditions on the resource being tested, but can work on VMs and physical machines alike) and using ESXi screenshot functionality (which only works for VMs, but works for all OS families and is unaffected by conditions on the resource being tested). Here, we cover how to setup and verify ESXi-based event detection.

This appending assumes that the test range you're using has already been set up for event detection.

5.1 Event Detection Theory

Event detection works by taking screenshots according to a configurable interval and comparing them to find differences. When differences are found, they are analyzed to determine whether they are considered significant or not. A difference is significant if the number of changed pixels in a set of predefined areas of interest exceeds a defined threshold. The areas of interest and the threshold were determined empirically and are defined in `leafbags/tyutils/event_detection/event_detectors/__init__.py` as percentage boxes bounding the areas of interest. The current threshold is 10000 pixels, and the current areas of interest are:

- A box in the bottom right corner of the screen, extending 25% toward the left and 50% toward the top.
- A box in the center of the screen, whose edges are all 30% away from their respective screen edge (i.e. the left edge of the box is 30% from the left edge of the screen, the bottom edge of the box is 30% from the bottom edge of the screen, etc).
- A box in the upper right corner extending 20% toward the left and 20% toward the bottom.

When no problems occur with event detection, the event detection harness simply returns the result returned by the underlying test script. If, however, the test script returns SUCCESS, but any problems are encountered with event detection (e.g. not being able to take screenshots frequently enough to satisfy the configured interval), then the harness will return `ATTENTION` along with a message describing what happened.

5.2 Testing in Adverse Environments

The goal of testing is always to determine truthful outcomes to potential scenarios as early as possible, so that risks can be understood and evaluated. It is critical for users and testers to be aware of the fact that testing of any kind produces traces and artifacts. These traces and artifacts are created because it is impossible to actuate components that set up test preconditions without changing the state of the machine under test. There are many ways to achieve these actuations. Different methods can (and

sometimes do) provide different results. This is especially true in adverse environments. For example: a tool that passes when a user runs the program from the desktop with the mouse could fail or cause a pop-up if started by another process.

Automated testing frameworks like DART allow testers to cover a greater number of potential scenarios than they could manually. This creates more confidence in the tools being tested. However, it is critical to understand that the automated framework runs in a formulaic way, so it is possible that the methods chosen could routinely produce different results in the real world. It is even possible that by allowing the automated framework to run in an adverse environment, that environment will be changed enough that a different result could show up.

Tester note: You should run a statistically relevant subset of the tests by hand to verify the results given by the automated framework. Follow the spirit of the test plan and ensure that doing things manually produces the same results. This will nearly always be the case, but we have observed instances where there are slight deviations in the past.

5.3 Environment Setup

In order to use event detection, you need to do some setup in your local testing environment (the environment in which you run `remote_commit` to submit tests to the range).

- In the same directory where your tyworkflow and tybase clones are, clone the tyutils repository (`hg clone http://testserver.example.com:8000/tyutils`).
- In that same directory, also clone the provided PIL (Python Imaging Library) repository matching the architecture of the test server. For example, if your test server is running 32-bit Linux, choose the "PIL-linux-i686" repository, but if it's running 64-bit Linux, use "PIL-linux-x86_64". This library is used to compare screenshots to find changes.
- In your tyutils clone, copy `config/main.conf.example` to `config/main.conf` and set the following settings. Tyutils has many features besides event detection, so some of these settings are unrelated but need to have some value set for them to prevent warning messages.
    - Set `tester/evdet_type` to `esxi`.
    - Set `tester/esxi_evdet_ds_name` to the name of the NFS datastore you created in the previous section (e.g. `tyrantshare`).
    - Set `tester/esxi_evdet_local_ds` to the path on the test server of the directory what was exported via NFS in the previous section (e.g. `/proj/testing/tyrantshare`).
    - Set `server/ip_addr` to `127.0.0.1`.

5.4 Usage

To use event detection, you use an event detection harness leafnode provided by tyutils. You tell it what test script you want it to run and what arguments and keyword arguments to run it with and give it various other pieces of information to configure the event detection. The harness takes the following arguments:

- `host_index`: Zero-counting integer index of which host event detection should be performed on. This is necessary when your test case uses more than one host, but you want event detection run on some host other than the first. Default is to use the first host.
- `interval`: Float number of seconds for the screen polling interval. Default is 5 seconds.
- `use_emissary`: Boolean indicating whether or not to use emissary when using the `onhost` event detection method. Not applicable for `esxi` event detection. Default is `False`.
- `type`: Selects which type of event detection to perform (`esxi` or `onhost`). Default is `onhost`. For what this appendix is covering, you will always choose `esxi` here.
- `esxi_host`: For `esxi` event detection, specifies the ESXi server to connect to to take screenshots. This may be either the specific ESXi host the VM resides on, or (we assume, but have not tested) a vCenter server for the range in which the VM resides (we assume this because other operations like reverting work both when directly connected to an ESXi host or when connected to a vCenter server). If not specified, the harness will attempt to query the overmind database (if present) and will assume the reaper field for the test resource indicates the DNS name or IP address of the ESXi host the VM resides on.
- `esxi_user`: Username to use for connecting to the ESXi host.
- `esxi_pass`: Password to use for connecting to the ESXi host.
- `vm_name`: ESXi name of the VM. If not specified, the harness will attempt to query the overmind database (if present) and will use the computer name field as the VM name in ESXi.
- `debug`: Boolean indicating whether or not to perform event detection debugging. This causes the generation of extra log messages and several intermediate images during the screen differencing process. Do not use this unless you actually need to debug the screen differencing process. This does not affect debugging for the test script being wrapped. Default is `False`.
- `debug_dir`: If `debug` is `True`, specifies a directory path to which to output the extra files generated by debugging. Default is to use a subdirectory of the output directory for the run of the harness.
- `keep`: Boolean indicating whether or not to keep screenshots which indicate differences determined to be insignificant. Default is `False`.
- `test`: Specification of the leafnode to run. Specify this as a "python import path", not a filesystem path.
- `test_args`: List of positional arguments to the specified leafnode.
- `test_kwargs`: Dict of keyword arguments to the specified leafnode.

Note that these arguments are subject to the argument quoting rules of undermine. See the output of `bin/undermine -h` (run in tybase) for details.

5.4.1 With Undermine

To use event detection to run single test instances with undermine, simply call the event detection harness with the proper arguments. For example:

```
      bin/undermine tyutils.event_detection.harness.evdet_harness
192.168.56.101  192.168.56.103 192.168.56.104 -- host_index=@1
interval=@3.0 type=esxi esxi_host=192.168.56.10 esxi_user=someuser
      esxi_pass=somepass vm_name=test_vm_001
```

```
test=mysut.tests.sometest test_args=@"[yes, yes, no]"
test_kwargs=@"{one=1, two=2}"
```

If running out of an environment linked to an overmind range (i.e. tybase and tyworkflow are linked and tyworkflow's `db.rc` is configured to use an overmind database) and using IP addresses that are part of the range, you can leave out the `esxi_host` and `vm_name` arguments and they will be determined automatically by looking in the overmind database.

5.4.2 With Overmind

To use event detection in an overmind range, for each test script you wish to run with event detection, you'll need to write your own test plan that calls the event detection harness with the arguments as explained previously.

To make this easier, start with the following template plan:

```
from tyworkflow.support.planlang import *

test = TESTCASE(
    script = 'tyutils.event_detection.harness.evdet_harness',
    hostslots = [PUT_HOSTS_HERE],
    samples = -1,
    namespace='TEST_NAME-$t',
    paramslots = [
        ['test=LEAFNODE_SPEC'],
        ['esxi_user=USERNAME'],
        ['esxi_pass=PASSWORD'],
        ['keep=@True']
    ]
)

EXECUTE(
    testcase = test,
)
```

Save this code to a new file with your other plans and scripts (DO NOT simply modify this file in place), and modify the copy as follows:

- In the `hostslots` parameter, replace PUT_HOSTS_HERE with HOST objects according to the number and type of hosts your test script requires. (If your test script takes multiple hosts and the host on which you want event detection is not the first host, remember to set the `host_index` keyword argument in `paramslots`).
- If you want to have a default limit on the number of samples the plan will generate, then change the samples parameter. You probably just want to leave it at `-1`, though.
- In the `namespace` parameter, replace TEST_NAME with a very concise name of your test; this will be used as part of the default namespace name when submitting this plan.
- In paramslots,

- Replace LEAFNODE_SPEC with the specification of your leafnode (using “python import path”, as before).
- Replace USERNAME and PASSWORD with the username and password used to log in to the ESXi hosts on your range.
- If you have a central vCenter server managing all the ESXi hosts in the range, you can define add the `esxi_host` parameter in `paramslots` with the vCenter server’s DNS-resolvable hostname or IP address as the value. Otherwise, leave `esxi_host` out and it will be determined by looking in the overmind database.
- Add definitions for `test_args` (a list) and `test_kwargs` (a dict) to `paramslots` to define the positional and keyword arguments to be passed to your test script being run under event detection.

At this point, you should now have a working test plan that will run your test script under event detection. You can run this test plan with remote commit like any other, and see its results in overview, except now, you will get screenshots as well.

5.5 vmwareScreenshot

Event detection essentially uses screenshots either on host or via VMWare’s screenshot capture. If you wish to take screenshots outside of event detection, you can call the `vmwareScreenshot` function from the tyutils leafbag. The following test sample below demonstrates taking a screenshot on a VMWare resource. This code can also be found in *tytils/leafbag/tests/screenshot_test.py*.

```
import tybase.undermine.meta.leafi as leafi
import tybase.undermine.main_script as main_script
from tyutils.resource_utils import vmwareScreenshot

import time

@leafi.MainLeaf()
class ScreenShotTest(main_script.Main_Script):
    def run(self):
        failed_hosts = []

        for host in self.hosts:
            host.log.info("About to take Screenshot on:", host.ip)
            rv = vmwareScreenshot(host)
            host.log.info("VMWare Screenshot return value:", rv)
            if rv == None:
                host.log.error("Failed to take screenshot on:", host.ip)
                failed_hosts.append(host.ip)

        if len(failed_hosts) > 0:
            msg = "Failed to take screenshots on: "+str(failed_hosts)
            return(self.FAILURE, msg)
        else:
            msg = "Done with test, please review logs to " + \
            "double check test passed"
            return(self.SUCCESS, msg)
```

6 Appendix B - Detailed Repository Layouts

1 tybase

- bin: Shell scripts used to run Tyrant components present in tybase
- docs: Some documentation which is superseded by this manual
- leafbags: Directory in which collections of test scripts and plans are linked in, making them available to be run by undermine or overmind
- media: Directory in which third-party supporting media is included or linked in
    - lib_esxi-0.1: library which allows controlling ESXi servers via the vSphere API
- PythonLocal: Built-in python distribution used by Tyrant. This directory is only present after running `make`.
- rc: Configuration files for components in tybase.
    - defaults: Default settings which are checked into the repository; these are overridden by settings in the files directly in rc.
- src: Source code for Tyrant components in tybase. This directory is present on the python path when running any Tyrant components.
    - leafbag: A collection of built-in leafnodes.
    - tybase
        - hal: Source code for the HAL component, used to perform computer-level operations on test resources
        - palantir: Source code for the palantir component, used to run operations on test resources
            - installer: The code used to install palantir on test resources, as well as a pre-built python for the OS/architecture combinations supported by tybase.
        - support: Supporting modules used by various parts of Tyrant.
        - undermine: Source code for parsing and running leafnodes.
- test: A collection of regression tests for tybase components.

2 tyworkflow

The tyworkflow repository is structured similarly to tybase. The list below highlights the differences.

- install: Code used to install overmind and reaper as a system service.
- src
    - leafbag: Built-in leafbag containing test scripts and plans for verifying a range is properly set up.
    - tyworkflow
        - overmind: Source code for overmind, the component which schedules tests across shared resources.
        - overview: Source code for the web gui used to see test results and manage a range.
        - overview_httpd: Built-in web server for running overview in small environments (e.g. on a laptop while traveling).

- reaper: Source code for the reaper component, which handles sanitizing resources prior to a test.
- resource_manager: Source code for the component which tracks the state of test resources using a database.
- support: Supporting code for various components of tyworkflow.

6.1 tyutils/leafbag

The tyutils repository contains various modules of testing utilities in its *leafbag* directory. The root of tyutils/leafbag directory contains various source code and modules. Source code may increase and improve over multiple Tyrant releases. The general structure is outlined below.

- tests: A collection of sample tests for tyutils APIs and utilities.
- *.py: Different utility modules and code files.
- event_detection: Source for event detection (formerly in magnum repository)
  - config: TYUTILS configuration files.
    - defaults: Default settings for tyutils event detection; overridden by the files directly in config.
  - event_detectors: Modules implementing the two types of event detection.
  - harness: A leafnode which can be used to wrap your own leafnode with event detection logic, as well as some code for testing event detection.
  - util: Common utility functions used by event detectors and harness scripts.

*3 PIL-**

These repositories each contain a PIL subdirectory in which reside the Python Imaging Library code and compiled components.

7 Appendix C – Commands and Usage

All of these usage statements can be found by typing command –h from the command line.

4 Tyworkflow

7.1.1 remote_commit

```
usage: bin/remote_commit [-u user] [-h|--help|help] [start|stop|restart|diff|
summary|build|

    clobber|rclobber|sync[lite]|run[lite] [plan] [opts]|

    (parse|plan|solve|submit) [plan] [opts]|autoplan [args]|

    auto(parse|solve|submit) [args]|autorun[lite] [args]|get_children|

    set_children #|get_popthreads|set_popthreads #|purge [(nid|pid|tid)=#]|

    client [args]]


Run operations against a remote overmind testing server set up to use the
remote

commit system.


-u allows you to specify a "remote commit username", which is used as the

subdirectory of the remote commit root on the remote server and to identify a

specific remote overmind instance. If not specified, the username defaults to

the username of the user running remote commit (i.e. $USER, meaning
nlsheppa@dart.local).


Remote commit commands generally use other tyrant components to do their work.

You may be interested in reading the output of the following commands for
extra

details:

    bin/process_plan -h

    bin/overmind_admin -h

    bin/autoplan -h
```

```
The meanings of the commands and their options are as follows:

    start: Start the remote overmind instance.
    stop: Stop the remote overmind instance.
    restart: Sestart the remote overmind instance.
    diff: Does a dry run of syncing your local environment up to the remote
        server.
    summary: Summarizes the test results in your remote environment's output
        directory.
    build: Locally builds your SUT according to the provided _rcoverride_build
        function.
    clobber: Locally clobbers your SUT according to the provided
        _rcoverride_clobber function.
    rclobber: Remotely clobbers your SUT according to the provided
        _rcoverride_clobber present in the remote enviroment as part of the
        files you synced over.
    sync|synclite: Syncs your local testing environment up to the server. The
        synclite command only syncs; the full sync command clobbers and then
        builds your SUT first (just like the build command) and restarts the
        remote overmind instance if it's currently running.
    run|runlite plan [opts]: Syncs your environment to the remote server, then
        submit the given plan. Parameters are the plan to run and any other
        options as would be appropriate for bin/process_plan.  Full run
clobbers
        and then builds your SUT before syncing, runlite does not.
    parse|plan|solve|submit plan [opts]: Runs the specified operation using
the
        given plan.  These four commands and any parameters have the same
```

meaning as with bin/process_plan.

autoplan *: Runs the automatic plan generator on the remote server. See bin/autoplan -h for details on what arguments it takes. This form does not handle the remote overmind instance for you. You should use the other auto* commands following this one, instead of this one. This command may someday be deprecated.

auto(parse|solve|submit) [args]: Shortcuts for autoplan commands run on the remote server. These behave just like the normal parse|solve|submit commands, but automatically generate a plan rather than running a specified existing plan.

autorun|autorunlite [args]: Syncs your environment to the remote server, then submits an autogenerated plan according to the commandline arguments. Parameters are those accepted by the bin/autoplan tool. Full autorun clobbers and then builds your SUT before syncing, autorunlite does not.

get_children: Returns the current setting for maximum number of undermine processes to run at once on the remote server.

set_children #: Sets the maximum number of undermine processes to run on the remote server.

get_popthreads: Returns the current setting for maximum number of post-op threads running on the remote server.

set_popthreads: Sets the maximum number of post-op threads to run on the remote server.

purge [(nid|pid|tid)=#]: Purges as plan from your remote overmind instance. Namespace id (nid), plan ID (pid) or (tid) may be specified, and all matching testcases will be purged. If no arguments are given, all

```
        testcases running in your remote overmind instance will be purged.

    client [args]: Runs the overmind client against the remote overmind
instance

        with whatever parameters you give it.
```

7.1.2 db_admin

```
Usage: db_admin [<option>]* <method> [arg]*


Options:

  -h           : Print help.

  -c <rc_file> : Use rc_file as an alternate config file.

  -v           : Turn on verbose diagnostic output. (logging.DEBUG)

  -q           : Turn off verbose diagnostic output. (logging.WARN)

  -j           : return JSON-encoded output


Methods defined here:


active_clone_status(self, clone_status_id) from
tyworkflow.resource_manager.client_util.DB


add_attr(self, attr_table, attr_name) from
tyworkflow.resource_manager.client_util.DB

    Adds a formal attribute of the given name attr_name (string) to the

    given table attr_table (string).


add_computer(self, computer, *args, **kwargs) from
tyworkflow.resource_manager.client_util.DB

    Adds a computer of the given name computer (string) with the

    attributes as specified in *args/**kwargs. Generic computer attributes

    and their order if specified positionally are as follows:

        formal attributes:
```

```
            ip: (str) IP address of primary interface

            mac: (str) MAC address of primary interface

            hwtype: (str) name of hardware type of computer

            model: (str) model of the computer

            pool: (str) name of pool in which to place computer

            vlan: (str) name of vlan in which to place computer

            state_type: (str) name of state control implementation to use for
the computer

            reaper: (str) name of reaper configuration to use for reaping the
computer



    Hardware-type-specific extended attributes must be specified as kwargs

    and consist of the following:

        for hwtype phys:

            pdu_host: (str) hostname/IP address of PDU computer is connected
to; required at import

            pdu_model: (str) model of PDU computer is connected to; read-only

            pdu_outlet: (str) outlet identifier for where computer is
connected to PDU; required at import

            pdu_user: (str) username to use to log into the PDU; read-only

            pdu_passwd: (str) password to use to log into the PDU; read-only

        for hwtype vm:

            vm_host: (str) hostname/IP of VM host the computer resides on;
required at import

            vm_type: (str) type of VM host computer resides on; read-only

            vm_host_user: (str) username to use to log into the host the
computer resides on; read-only

            vm_host_passwd: (str) password to use to log into the host the
computer resides on; read-only

            vm_host_max_in_use: (int) maximum number of VMs concurrently in
use by automated tests on the VM host this VM resides on; <= 0 or NULL means
no limit
```

```
    Returns: the ID of the inserted computer entry


add_dbuser(self, name, level, password=None) from
tyworkflow.resource_manager.client_util.DB

    Creates a new database user at the given privilege level. This

    functionality is specific to the MySQL database engine.


    Parameters:

        str name: name of the new user

        str level: level identifier, one of the following:

            view: read-only access to the database; allows one to view test
results but not run tests

            test: allows one to execute tests and perform other operations a
tester might want to do

            admin: gives full control of the database

        str password: password for the user; if not specified, will be read

            from standard input


    Returns: True on success

    Raises: StandardError on error


add_pdu(self, *args, **kwargs) from tyworkflow.resource_manager.client_util.DB

    Adds a PDU with the given name (which must be the hostname or IP

    address at which the PDU may be accessed) and other attributes as

    specified in *args/**kwargs. Attributes and their order if specified

    positionally are as follows:

        metadata attributes:

            host: (str) hostname/IP address to connect to the PDU
```

```
            model: (str) model of the PDU

            user: (str) username used to log in to the PDU

            passwd: (str) password used to log in to the PDU


    Returns: the ID of the inserted PDU entry


add_recipe(self, recipe, *args, **kwargs) from
tyworkflow.resource_manager.client_util.DB

    Adds a recipe of the given name recipe (string) with the attributes

    as specified in *args/**kwargs. Valid attributes and their order if
specified

    positionally are as follows:

        formal attributes:

            family: (str) OS family enumeration (e.g. 'win', 'linux')

            os: (str) OS name (e.g. 'xp', 'vista', 'fedora')

            ossp: (str) OS service pack designation

            lang: (str) OS language enumeration (e.g. 'en-US')

            arch: (str) OS architecture enumeration, typically either 'x86' or
'x86_64'

            apps: (str) names of apps installed in the recipe


    Returns: the ID of the inserted recipe entry


add_snapshot(self, computer, recipe, snapshot=None) from
tyworkflow.resource_manager.client_util.DB

    Adds a snapshot with a certain recipe to a computer.


    Parameters:

        str|int computer: name or ID of the computer to add the snapshot to

        str|int recipe: name or ID of the recipe of the snapshot being added
```

```
        str snapshot: name of the snapshot being added, defaults to latest if
not

            set



    Returns: the ID of the added snapshot



add_vlan(self, name, ip_min, ip_max, mac_min, mac_max) from
tyworkflow.resource_manager.client_util.DB

    Adds a vlan with the given attributes. Attributes are as follows:

        metadata attributes:

            name: (str) name of the vlan; where applicable, should match the
name of the network to which the computer's network interface is connected
(e.g. the network name on an ESXi host)

            ip_min: (str) minimum IP address bounding a pool of IP addresses
from which pick a unique one when needed (e.g. cloning)

            ip_max: (str) maximum IP address bounding a pool of IP addresses
from which pick a unique one when needed (e.g. cloning)

            mac_min: (str) minimum MAC address bounding a pool of MAC
addresses from which pick a unique one when needed (e.g. cloning)

            mac_max: (str) maximum MAC address bounding a pool of MAC
addresses from which pick a unique one when needed (e.g. cloning)



    Returns: the ID of the inserted vlan entry



    Raises: ValueError if given MAC or IP addresses are in an invalid

    format.



add_vm_host(self, *args, **kwargs) from
tyworkflow.resource_manager.client_util.DB

    Adds a VM host with the given attributes (as specified in

    *args/**kwargs. Attributes and their order if specified positionally are

    as follows:
```

```
        metadata attributes:

            host: (str) hostname/IP address to connect to the VM host

            type: (str) type of VM host (e.g. esxi, vbox, etc)

            user: (str) username used to log in to the VM host

            passwd: (str) password used to log in to the VM host

            max_in_use: (int) maximum number of VMs in use concurrently for
automated tests on this host; set <= 0 or NULL for no limit



    Returns: the ID of the inserted VM host entry



atomic_add_attr(self, attr_table, attr_name) from
tyworkflow.resource_manager.client_util.DB

    Atomically creates the given attribute, failing if the attribute

    already exists.



    Parameters:

        str attr_table: the attribute table the attribute is being created

            in (e.g. the type of attribute: mac, ip, etc)

        str attr_name: the attribute to create (e.g. the actual MAC address,

            IP address, etc)



    Returns: int ID of the attribute created, or None if the attribute

        already exists



atomic_reserve_asset(self, asset_name, reserve_name='') from
tyworkflow.resource_manager.client_util.DB

    Reserves a single asset (computer) atomically. If the computer is

    currently reserved or in use for a test, raises an Exception, otherwise

    returns True.
```

```
    Parameters:
        str asset_name: name of the computer (matching the DB's computer
            record)
        str reserve_name: name to store on the computer record to identify
            who's reserved it

    Returns: True upon successful reservation of the computer
    Raise: Exception if the computer is already reserved or an error
        occurs (such as the named computer not existing)
```

cleanup_clone(self, clone_status_id) from tyworkflow.resource_manager.client_util.DB

condense_contentions(self, interval=900, level=0, epoch=None, hold=0) from tyworkflow.resource_manager.client_util.DB

Condenses contentions metric data. For example, with default parameters, data having idential resource ids, reason, and

within the same 15 min interval from now (and have not yet been condensed) will be summed and replaced with one record.

Optional parameters:

interval: (int) interval in seconds to condense (defaults to 900 sec, or 15 min)

level: (int) level of condension. Will condense all data at or less than level and mark the condensed record with level+1

Records at level 0 have not yet been condensed. This is the default level.

epoch: (int) Datetime epoch of latest records to start condensing. Default is currrent timestamp

hold: (int) Number of intervals to hold back on condensing. Default is 0.

For example, with hold=1 and all other default parameters, and this function is called at 10:15, only data older than 10:00

will be condensed.

del_attr(self, attr_table, attr_name) from tyworkflow.resource_manager.client_util.DB

Deletes the attribute attr_name (string) from the table attr_table

(string). Returns the entry which was deleted, or None if the attribute didn't

exist.

del_computer(self, computer) from tyworkflow.resource_manager.client_util.DB

Deletes the named computer (string). Returns the record of

the deleted computer, or None if the given computer didn't exist.

del_contentions(self, age, epoch=None) from tyworkflow.resource_manager.client_util.DB

Deletes contention records for data older than given age (in days) starting from given epoch.

Default start is current timestamp.

For example, if age is 3, all contentions older than three days from now will be deleted.

To purge all contentions, set age to 0.

Required parameters:

age: (int) number of days older than given epoch to delete

Optional parameters:

epoch: (int) Datetime epoch to calculate age for deletion. Default is currrent timestamp

```
del_dbuser(self, name) from tyworkflow.resource_manager.client_util.DB

    Deletes the specified database user. Returns True on success. Raises

    Exception on error.


del_pdu(self, host) from tyworkflow.resource_manager.client_util.DB

    Deletes the PDU entry with the given host field value.


del_recipe(self, recipe) from tyworkflow.resource_manager.client_util.DB

    Deletes the named recipe (string). Returns the record of the deleted

    recipe, or None if no such recipe existed.


del_reservation_history(self, age, epoch=None) from
tyworkflow.resource_manager.client_util.DB

    Deletes reservation history data older than given age (in days) starting
from given epoch.

    Default start is current timestamp.

    For example, if age is 3, all history older than three days from now will
be deleted.

    To purge all history, set age to 0.



    Required parameters:

      age: (int) number of days older than given epoch to delete



    Optional parameters:

      epoch: (int) Datetime epoch to calculate age for deletion. Default is
currrent timestamp


del_snapshot(self, computer, recipe=None, snapshot=None) from
tyworkflow.resource_manager.client_util.DB
```

```
    Deletes snapshots from the named computer (string). The recipe and
    snapshot arguments (strings), if provided, are used to filter the
    snapshots that get deleted (otherwise, all snapshots for the specified
    computer are deleted.


del_vlan(self, name) from tyworkflow.resource_manager.client_util.DB
    Deletes the vlan with the given name.


del_vm_host(self, host) from tyworkflow.resource_manager.client_util.DB
    Deletes the VM host with the given host field value.


describe_table(self, table) from tyworkflow.resource_manager.client_util.DB
    Returns a string description of the table named table (string).


drop_db(self, *args) from tyworkflow.resource_manager.client_util.DB
    Drop all tyrant tables and entries including test results.
    The argument list must include the magic string "+really-do-it" to help
prevent
    accidental execution of this command, or optionally, you will be prompted
for
    confirmation prior to command execution.


drop_resources(self, *args) from tyworkflow.resource_manager.client_util.DB
    Drop the tyrant resource tables and entries (computer, recipe, snapshot).
    The argument list must include the magic string "+really-do-it" to help
prevent
    accidental execution of this command, or optionally, you will be prompted
for
    confirmation prior to command execution.
```

```
drop_tests(self, *args) from tyworkflow.resource_manager.client_util.DB

    Drop the tyrant test result tables and entries.

    The argument list must include the magic string "+really-do-it" to help
prevent

    accidental execution of this command, or optionally, you will be prompted
for

    confirmation prior to command execution.



dump_table(self, table) from tyworkflow.resource_manager.client_util.DB

    Returns all the records in the given table (string).



dump_version(self) from tyworkflow.resource_manager.client_util.DB

    Returns the version of the database.



export_computers(self, csvfile='-') from
tyworkflow.resource_manager.client_util.DB

    Export computers to csv file (or stdout), format:

        <name>,<ip>,<mac>,<hwtype>,<pool>,<vlan>,<reaper>

        @snapshot,<recipe>,<snapshot>

    If the csv file is '-' (the default), write to stdin.



export_pdus(self, csvfile='-') from tyworkflow.resource_manager.client_util.DB

    Export snapshots to csv file (or stdout), format:

        <host>,<type>,<username>,<password>

    If the csv file is '-' (the default), write to stdout.



export_recipes(self, csvfile='-') from
tyworkflow.resource_manager.client_util.DB

    Export recipes to csv file (or stdout), format:

        <name>,<family>,<os>,<ossp>,<lang>,<arch>,<apps>
```

```
    If the csv file is '-' (the default), write to stdin.


export_snapshots(self, csvfile='-') from
tyworkflow.resource_manager.client_util.DB

    Export snapshots to csv file (or stdout), format:

        <computer|id>,<recipe|id>,<snapshot|id>

    If the csv file is '-' (the default), write to stdout.


export_vlans(self, csvfile='-') from
tyworkflow.resource_manager.client_util.DB

    Export vlans to csv file (or stdout), format:

        <name>,<ip_min>,<ip_max>,<mac_min>,<mac_max>

    If the csv file is '-' (the default), write to stdout.


export_vm_hosts(self, csvfile='-') from
tyworkflow.resource_manager.client_util.DB

    Export snapshots to csv file (or stdout), format:

        <host>,<type>,<username>,<password>

    If the csv file is '-' (the default), write to stdout.


get_attr_id(self, attr_table, attr_name) from
tyworkflow.resource_manager.client_util.DB

    Returns the ID for an attribute.


    Parameters:

        str attr_table: name of the table to look for the attribute in

        str attr_name: name (or possibly ID [but still must be a string, not

            an int]) of the attribute to look for


    Returns: the ID of the attribute found
```

```
get_clone_lock(self, clone_status_id) from
tyworkflow.resource_manager.client_util.DB

    # CLONE
----------------------------------------------------------------------



get_clone_status(self) from tyworkflow.resource_manager.client_util.DB



import_computers(self, csvfile='-', testing_use='N', testing_dirty='Y',
**kwargs) from tyworkflow.resource_manager.client_util.DB

    Import computers from csv file (or stdin), format:


<name>,<ip>,<mac>,<hwtype>,<model>,<pool>,<vlan>,<state_type>,<reaper>,
[extended_attrs ...]

        @snapshot,<recipe>,<snapshot>

    If the csv file is '-' (the default), read from stdout.



    The extended_attrs part is for specifying hwtype_specific attributes at

    import time. Like the rest of the fields in a CSV input line, these are

    comma-separated values, but they are given as key/value pairs.  They key

    specifies the extended attribute name, and the value is the value to set

    for it. For example, if you have a computer which has hwtype == vm, you

    could set the "vm_host" attribute to the name of the host the VM resides

    on (matching the VM host name you import with import_vm_hosts), and your

    imported computer record would be linked to the vm_host table entry for

    the specified VM host.  For example, you could do:

        00-3c-
test_comp_001,192.168.5.50,00:50:56:00:3c:00,vm,esxi,pool001,vlan001,esxi,myre
aper,vm_host=my_esxi_host

    Similarly, for a computer with hwtype == phys, you could set pdu_host to

    the hostname of a PDU imported with import_pdus, and pdu_outlet to the
```

```
    name (usually just a numerical index) of the outlet on the PDU to which
    the computer is connected.

    Note that for attributes like the above examples where the attribute you
    set results in a link from a computer to some other entity (such as a VM
    host or PDU), the entity being linked to (the VM host or the PDU) MUST
    already exist in the database BEFORE running import_computers.
    import_computer WILL NOT automatically create the entity being linked to
    for you (it cannot, since it doesn't have enough information to do so).

    Parameters:
        str csvfile: path to CSV file to import; if "-", read from standard
            input instead
        enum(Y|N) testing_use: value to set testing_use flag to on
            newly-imported computers; default is to mark computer not to be
            used for testing
        enum(Y|N|R) testing_dirty: value to set testing_dirty flag to on
            newly-imported computers; default is to mark computer as dirty
        **kwargs: keyword arguments to override values of some computer fields

    Fields are defined in the list_computers method's help. The kwargs can
    define default values for all the formal attributes except id. The fields
    listed as "extended hwtype-specific attributes" are the ones which must
    be specified as key/value pairs at the end of a computer CSV line (the
    "extended_attrs").

import_pdus(self, csvfile='-') from tyworkflow.resource_manager.client_util.DB
```

```
    Imports PDUs from a CSV file.


    Format is:

        <hostname_or_ip>,<model>,<username>,<password>


    The "model" value dictates which PDU control implementation is used for

    computers associated with this PDU.  Thus, this field's value needs to

    match one of the implementations in

    tybase/src/tybase/hal/hwctl/phys/pdu.


import_recipes(self, csvfile='-', **kwargs) from
tyworkflow.resource_manager.client_util.DB

    Import recipes from csv file (or stdin), format:

        <name>,<family>,<os>,<ossp>,<lang>,<arch>,<apps>

    If the csv file is '-' (the default), read from stdin.

    The kwargs can define default values, for example: lang=en arch=x86


import_snapshots(self, csvfile='-', **kwargs) from
tyworkflow.resource_manager.client_util.DB

    Import snapshots from csv file (or stdin), format:

        <computer|id>,<recipe|id>,<snapshot|id>

    If the csv file is '-' (the default), read from stdin.

    The kwargs can define default values for the following parameters:


    str computer: name of the computer the snapshot is being added for

    str recipe: name of the recipe in the snapshot

    str snapshot: name of the snapshot
```

```
import_vlans(self, csvfile='-', **kwargs) from
tyworkflow.resource_manager.client_util.DB

    Import vlans from csv file (or stdin), format:

        <name>,<ip_min>,<ip_max>,<mac_min>,<mac_max>


    The fields have the following meanings:

        name: name of the VLAN. In some cases, this is merely a label, but
            where applicable (e.g. for VMs), this must match the name of the
            network to which the VM's network interface is connected. This
            is essential for clone support to work properly.
        ip_min/max: Minimum and maximum IP addresses which bound a pool of
            IP addresses used on this vlan. In scenarios where a new IP
            address needs to be assigned (e.g. cloning), it will be chosen
            from within these bounds (subject to other checks to ensure it's
            not in use). These MUST be IPv4 addresses in dotted-decimal
            notation; NO OTHER types of address or format of address
            representation are supported.
        mac_min/max: Minimum and maximum MAC addresses which bound a pool of
            MAC addresses used on this vlan. Used similarly to the
            ip_min/max. WARNING: The MAC MUST be specified with colons (':')
            as separators, NOT dashes ('-').

        The ip_min/max and mac_min/max fields MUST be specified.

        If the csv file is '-' (the default), read from stdin.
        The kwargs can define default values for any field of the CSV
        lines.
```

```
import_vm_hosts(self, csvfile='-') from
tyworkflow.resource_manager.client_util.DB

    Imports VM hosts from a CSV file.



    Format is:

        <hostname_or_ip>,<type>,<username>,<password>,<max_vms_in_use>



    The "type" value dictates which computer operations implementation will

    be used to perform computer operations against computer entities

    associated with this VM host. Thus, this field's value needs to match

    one of the implementations in tybase/src/tybase/hal/hwctl/vm.



    The "max_vms_in_use" field specifies how many VMs on this host may be in

    use for automated tests at any one time.



init_db(self, *args) from tyworkflow.resource_manager.client_util.DB

    Incoming format: [+really-do-it] [+drop]



    Create any missing required tyrant database tables incuding recipes,
computers,

    and test results. Optionally, if the argument list includes the string

    "+drop" the database is removed prior to the creation of new tables.  The
argument

    list must include the magic string "+really-do-it" to help prevent
accidental

    execution of this command, or optionally, you will be prompted for
confirmation

    prior to command execution.
```

```
list_attr(self, attr_table, with_header=False, filter_by=None, sort_by=None,
display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists attributes of rows from the given table, with the set of returned

    attributes modified by the given parameters. Ignore the limit_to

    parameter below.



    Parameters:

        str attr_table: name of the table to list attributes from

        bool with_header: If True, a header giving the names of the fields
will be

            output with the returned entries.

        str filter_by: A list in the form of a comma-separated string of
filtering

            rules. See the FILTERING section. Fields which may be filtered on
are

            listed in this object's docstring.

        str sort_by: A list in the form of a comma-separated string of sorting

            rules. Sorting rules are simply names of fields, optionally
prepended

            by '!' to indicate a reverse sort on that field. Default sort
order is

            ascending.

        str|int limit_to: Limit the return from the database to the given
number of

            records. This limit is applied at query time. NOTE: Not all list

            functions support this parameter. Check the parameter list to see
if it

            applies.

        str|int display_num: First index number (zero-counting) of rows to
return. Only

            those rows on or after the given index number will be returned.
Useful for
```

```
            paging results, perhaps.

        int cut_to: Truncates each row of results to the specified number of
fields.

        str select: A list in the form of a comma-separated string of field
names

            to select (as with the SQL SELECT clause). Only those fields will
be

            returned. Valid fields are the same fields which may be filtered,
and

            are listed in this object's docstring.


        Examples:

            list all resources running windows

                bin/db_admin list_resources filter_by="family=win"

            list all computers, sorted by ip, ascending

                bin/db_admin list_computers sort_by=ip

            list the ten recipes starting with recipe 2, sorted descending by
os

                bin/db_admin list_recipes display_num=2 limit_to=10 sort_by="!
os"

            see only the ip and mac addresses for all computers, without a
header row

                bin/db_admin list_computers select=ip,mac with_header=false


    FILTERING

    Filters are specified using one of the following operators:

        a ~= b: True if str a is matched by regex b

        a !~= b: False if str a is matched by regex b

        a == b: True if a equals b

        a <> b: True if a does not equal b

        a != b: True if a does not equal b
```

```
        a >= b: True if int(a) is greater than or equal to int(b)

        a <= b: True if int(a) is less than or equal to int(b)

         a > b: True if int(a) is greater than int(b)

         a < b: True if int(b) is less than int(b)

         a = b: True if a equals b



    Examples:

        for listing resources, match all resources with service pack 2 or
greater

            ossp >= 2

        for listing computers, list the computer with ip address 127.0.0.1

            ip == 127.0.0.1

        for listing test namespaces, get any namespaces starting with "jdoe"

            name ~= ^jdoe



list_clone_status(self, with_header=False, filter_by=None, sort_by=None,
display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Displays the status of clone jobs in progress and in queue.







    Returns: A table with the operation ID, operation (create, destroy,
elevate), source name (c_name), clone name (n_name), snapshot, reserve name,
in_progress



list_computers(self, with_header=False, filter_by=None, sort_by=None,
display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists computers according to the given parameters.

    See list_attr help for detailed parameter descriptions. Valid fields to
```

```
    select/filter are:

        formal attributes:

            id: (int) ID of this thing

            name: (str) name of this thing

            ip: (str) IP address of primary interface

            mac: (str) MAC address of primary interface

            hwtype: (str) name of hardware type of computer

            model: (str) model of the computer

            pool: (str) name of pool in which to place computer

            vlan: (str) name of vlan in which to place computer

            state_type: (str) name of state control implementation to use for
the computer

            reaper: (str) name of reaper configuration to use for reaping the
computer

        metadata attributes:

            testing_use: enum(Y,N) whether computer can be scheduled for
testing

            testing_in_use: enum(Y,N) whether computer is currently in use on
a test

            testing_dirty: enum(Y,N,R) whether computer is dirty (Y), clean
(N), or should be reaped (R)

            is_clone: enum(Y,N) whether computer is a clone

            reserve_name: (str) name given when reserving/scheduling the
computer

            reserve_time: (datetime) time at which machine was
reserved/scheduled



    Extended hwtype-specific attributes are:

        for hwtype phys:

            pdu_host: (str) hostname/IP address of PDU computer is connected
to; required at import

            pdu_model: (str) model of PDU computer is connected to; read-only
```

pdu_outlet: (str) outlet identifier for where computer is connected to PDU; required at import

pdu_user: (str) username to use to log into the PDU; read-only

pdu_passwd: (str) password to use to log into the PDU; read-only

for hwtype vm:

vm_host: (str) hostname/IP of VM host the computer resides on; required at import

vm_type: (str) type of VM host computer resides on; read-only

vm_host_user: (str) username to use to log into the host the computer resides on; read-only

vm_host_passwd: (str) password to use to log into the host the computer resides on; read-only

vm_host_max_in_use: (int) maximum number of VMs concurrently in use by automated tests on the VM host this VM resides on; <= 0 or NULL means no limit

list_contention_intervals(self, start=None, end=None, small_interval='15', with_header=True, filter_by=None, sort_by=None, display_num=None, cut_to=None, select=None) from tyworkflow.resource_manager.client_util.DB

Shows metric values for reasons of resources in contention within intervals start and end epoch time specification.

Every interval is listed from start to end given the small interval. If there are more than one unique attr_name/attr_val pairs

if a hit count does not exist for and interval for a name/val pair but exists for another name/val pair, it should be assumed

that the hit count for the missing interval for the name/val pair is the same as the hit count for the previous interval.

If no hit count exists for an interval for any name/val pairs, then a row still exists for that interval with hit count zero.

Optional parameters:

start: (int) epoch integer of latest start time of intervals. Default to current epoch.

start: (int) epoch integer of end time. Default to 3 days before start

small_interval: (int) small interval span in minutes. Defaults to 15 min.

Valid fields to select/filter are:

formal attributes:

attr_name: (str) recipe or computer attribute name (family, os, etc)

attr_val: (str) attribute value (windows, xp, etc)

roll_lev: (int) contention condense level

list_contentions(self, start=0, end=32976997200, group_by='attr_name,attr_val', with_header=True, filter_by=None, sort_by=None, limit_to=None, display_num=None, cut_to=None, select=None) from tyworkflow.resource_manager.client_util.DB

list_dbusers(self, with_header=False, filter_by=None, sort_by=None, display_num=None, cut_to=None, select=None) from tyworkflow.resource_manager.client_util.DB

Lists the users which currently exist. This command only lists those users which, based on tests this method performs, are determined to have been created by the add_dbuser command. Accounts not created by this command will not be shown, or in rare instances, may be shown but misclassified as to their privilege level.

See list_attr help for detailed parameter descriptions. Valid fields to select/filter are: name, level

list_namespaces(self, with_header=False, filter_by=None, sort_by=None, limit_to=None, display_num=None, cut_to=None, select=None) from tyworkflow.resource_manager.client_util.DB

Lists namespaces. See list_attr for detailed parameter descriptions. Valid fields to select/filter are:

formal attributes:

```
            id: (int) ID of this thing

            name: (str) name of this thing

            nid: (int) id of the namespace

            n_name: (str) name of the namespace

            start_time: (datetime) time namespace started running tests

            end_time: (datetime) time namespace finished running all tests

            combos_total: (int) number of combos in namespace

            success: (int) number of combos with success status

            failure: (int) number of combos with failure status

            attention: (int) number of combos with attention status

            skipped: (int) number of combos with skipped status

            error: (int) number of combos with error status

            purged: (int) number of combos with purged status

            running: (int) number of combos with running status

            pending: (int) number of combos with pending status

            n_notes: (str) notes for the namespace

            pid: (int) id of the testplan

            keywords: [(str)] keywords associated with the namespace


list_pdus(self, with_header=False, filter_by=None, sort_by=None,
display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists PDUs. See list_attr for detailed parameter descriptions. Valid

    fields to select/filter are:

        formal attributes:

            id: (int) ID of this thing

        metadata attributes:

            host: (str) hostname/IP address to connect to the PDU

            model: (str) model of the PDU
```

```
            user: (str) username used to log in to the PDU

            passwd: (str) password used to log in to the PDU


list_recipes(self, with_header=False, filter_by=None, sort_by=None,
display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists recipes. See list_attr for detailed parameter

    descriptions. Valid fields to select/filter are:

        formal attributes:

            id: (int) ID of this thing

            name: (str) name of this thing

            family: (str) OS family enumeration (e.g. 'win', 'linux')

            os: (str) OS name (e.g. 'xp', 'vista', 'fedora')

            ossp: (str) OS service pack designation

            lang: (str) OS language enumeration (e.g. 'en-US')

            arch: (str) OS architecture enumeration, typically either 'x86' or
'x86_64'

            apps: (str) names of apps installed in the recipe


list_reservation_history(self, with_header=True, filter_by=None, sort_by=None,
limit_to=None, display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists computers reservation history according to the given parameters.

    See list_attr help for detailed parameter descriptions. Valid fields to

    select/filter are:

        formal attributes:

            hid: (int) id of reservation history table

            computer_name: (str) computer name

            computer_id: (int) ID of reserved computer

            reserve_name: (datetime) time at which machine was reserved
```

start_time: (str) name given when reserving the computer

end_time: (datetime) time at which machine was unreserved

list_reservation_history_metrics(self, start=0, end=32976997200, group_by='computer_id', with_header=True, filter_by=None, sort_by=None, limit_to=None, display_num=None, cut_to=None, select=None) from tyworkflow.resource_manager.client_util.DB

Shows metric values for reservation history for computers within intervals start and end epoch time specification.

Optional parameters:

start: (int) epoch integer of earliest start time of all results

end: (int) epoch integer of latest end time of all results

Valid fields to select:

formal attributes:

sum: (int) sum metric

average: (int) average metric

minimum: (int) minimum metric

maximum: (int) maximum metric

computer_id: (int) computer id

name: (str) computer name

Valid fields to filter are 'computer_id' and 'name'

NOTE: when group by reserve_name only, SUM is misleading - and does not count overlapping reserve times.

list_resource_usage(self, group_by='name', start=0, end=32976997200, with_header=True, filter_by=None, sort_by=None, limit_to=None, display_num=None, cut_to=None, select=None) from tyworkflow.resource_manager.client_util.DB

```
    Shows metric values for resource usage within given start and end epoch
time specification.



    Optional Parameters:

        group_by: (str) comma separated list of resource attributes to group
metric results

        start: (int) epoch integer of earliest start time of all results

        end: (int) epoch integer of latest end time of all results



    Valid fields to select/filter are:

    formal attributes:

        sum: (int) sum metric

        average: (int) average metric

        minimum: (int) minimum metric

        maximum: (int) maximum metric

        family: (str) OS family enumeration (e.g. 'win', 'linux')

        os: (str) OS name (e.g. 'xp', 'vista', 'fedora')

        ossp: (str) OS service pack designation

        lang: (str) OS language enumeration (e.g. 'en-US')

        arch: (str) OS architecture enumeration, typically either 'x86' or
'x86_64'

        apps: (str) names of apps installed in the recipe

        name: (str) computer name

        rid: (int) testcase resource id

        start_time: (int) start epoch value

        end_time: (int) end epoch value



    Valid fields to group_by are:

    name, family, os, ossp, lang, arch, apps
```

```
list_resource_usage_recipe(self, group_by, start=0, end=32976997200,
with_header=True, filter_by=None, sort_by=None, limit_to=None,
display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Shows metric values for recipe usage within given start and end epoch time
specification.

    This function differs from list_resource_usage in that metric results are
grouped by one or more

    resource attributes, resulting in a sum of non-overlapping time spans. The
resulting columns

    start_time, end_time, average, minimum, and maximum will always return 0
or '-'.



    Required parameters:

      group_by: (str) comma separated list of recipe attributes for metric
results

                Valid fields to group by are:

                family, os, ossp, lang, arch, apps,

                hwtype, model, pool, vlan, state_type, reaper,

                pdu_host, pdu_model, vm_host, vm_type



    Optional Parameters:

        start: (int) epoch integer of earliest start time of all results

        end: (int) epoch integer of latest end time of all results



    Valid fields to select/filter are:

    formal attributes:

        sum: (int) sum metric

        average: (int) average metric

        minimum: (int) minimum metric

        maximum: (int) maximum metric
```

```
        family: (str) OS family enumeration (e.g. 'win', 'linux')

        os: (str) OS name (e.g. 'xp', 'vista', 'fedora')

        ossp: (str) OS service pack designation

        lang: (str) OS language enumeration (e.g. 'en-US')

        arch: (str) OS architecture enumeration, typically either 'x86' or
'x86_64'

        apps: (str) names of apps installed in the recipe

        name: (str) computer name

        start_time: (int) start epoch value

        end_time: (int) end epoch value


list_resources(self, with_header=False, filter_by=None, sort_by=None,
display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists resources. See list_attr for detailed parameter

    descriptions. Valid fields to select/filter are any of the fields for

    computer, recipe or snapshot (see the corresponding list_* methods for

    the lists of valid fields), except that the "id" fields for each are

    called "computer_id", "recipe_id" and "snapshot_id", respectively, and

    the "name" fields are called "computer", "recipe", and "snapshot",

    respectively.


list_snapshots(self, with_header=False, filter_by=None, sort_by=None,
display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists snapshots. See list_attr for detailed parameter

    descriptions. Valid fields to select/filter are:

        formal attributes:

            id: (int) ID of this thing

            name: (str) name of this thing
```

```
            computer: (str) name of computer the snapshot is on, or specify as
computer_id and integer ID of computer

            recipe: (str) name of recipe on the snapshot, or specify as
recipe_id and integer ID of recipe

            snapshot: (str) name of snapshot

        metadata attributes:

            testing_fubar: enum(Y,N) whether or not the snapshot is broken



list_tables(self, with_header=False) from
tyworkflow.resource_manager.client_util.DB

    Lists tables in the database. If with_header is True, a header

    will be returned with the results (which in this case is just the string
'name'

    since the results are single-field rows of table names).



list_test_resource_mapping(self, with_header=False, filter_by=None,
sort_by=None, limit_to=None, display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists testcases resource mapping. See list_attr help for detailed
parameter

    descriptions. The limit_to parameter is valid for this method. Valid

    fields to select/filter on are:



list_testcase_files(self, with_header=False, filter_by=None, sort_by=None,
display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists files in all the testcases. See list_attr for detailed

    parameter descriptions. Valid fields to select/filter on are:

        formal attributes:

            tid: (int) id of the testcase the file is for

            id: (int) id of the file

            output_path: (str) path to the file
```

```
    In addition, a "name" field (str) may be used only for filtering on,
    which sometimes contains a relative path to the file.


list_testcase_resources(self, with_header=False, filter_by=None, sort_by=None,
limit_to=None, display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB
    Lists testcase resources according to the given parameters.
    See list_attr help for detailed parameter descriptions. Valid fields to
    select/filter are:
        metadata attributes:
            name: (str) computer name
            ip: (str) IP address of primary interface
            mac: (str) MAC address of primary interface
            hwtype: (str) name of hardware type of computer
            model: (str) model of the computer
            pool: (str) name of pool in which to place computer
            vlan: (str) name of vlan in which to place computer
            state_type: (str) name of state control implementation to use for
the computer
            reaper: (str) name of reaper configuration to use for reaping the
computer
            family: (str) OS family enumeration (e.g. 'win', 'linux')
            os: (str) OS name (e.g. 'xp', 'vista', 'fedora')
            ossp: (str) OS service pack designation
            lang: (str) OS language enumeration (e.g. 'en-US')
            arch: (str) OS architecture enumeration, typically either 'x86' or
'x86_64'
            apps: (str) names of apps installed in the recipe
            pdu_host: (str) hostname/IP address to connect to the PDU
            pdu_model: (str) model of the PDU
```

```
            pdu_outlet: (str) outlet for computer on the PDU

            vm_host: (str) hostname/IP address to connect to the VM host

            vm_type: (str) type of VM host (e.g. esxi, vbox, etc)

            combos_total: (str) total number of combos run on this resource

            combos_success: (int) number of combos with success status

            combos_failure: (int) number of combos with failure status

            combos_attention: (int) number of combos with attention status

            combos_skipped: (int) number of combos with skipped status

            combos_error: (int) number of combos with error status

            combos_purged: (int) number of combos with purged status

            combos_running: (int) number of combos with running status

            combos_pending: (int) number of combos with pending status


list_testcases(self, with_header=False, filter_by=None, sort_by=None,
limit_to=None, display_num=None, cut_to=10, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists testcases. See list_attr help for detailed parameter

    descriptions. The limit_to parameter is valid for this method. Valid

    fields to select/filter on are:

        formal attributes:

            tid: (int) id of the testcase

            t_name: (str) name of the testcase

            pid: (int) id of the testplan the testcase is in

            p_name: (str) name of the testplan the testcase is in

            nid: (int) id of the namespace the testcase is in

            n_name: (str) name of the namespace the testcase is in

            s_name: (str) name of the script the testcase is running

        metadata attributes:

            start_time: (datetime) time testcase started running
```

```
            end_time: (datetime) time testcase ended running

            result_code: (str) result code testcase ended with

            result: (str) result value testcase ended with

            result_code_orig: (str) original result code if manually changed

            result_orig: (str) original result value if manually changed

            change_comment: (str) comment for manual result change

            result_change_time: (datetime) time testcase result manually
changed

            presetup: (str) presetup script condition

            postcleanup: (str) postcleanup script condition

            o_path: (str) path of output directory for testcase


list_testplans(self, with_header=False, filter_by=None, sort_by=None,
limit_to=None, display_num=None, cut_to=None, select=None) from
tyworkflow.resource_manager.client_util.DB

    Lists testplans. See list_attr help for detailed parameter

    descriptions. The limit_to parameter is valid for this method. Valid

    fields to select/filter on are:

        formal attributes:

            pid: (int) id of the testplan

            p_name: (str) name of the testplan

            nid: (int) id of the namespace the testplan is in

            n_name: (str) name of namespace the testplan is in

            s_name: (str) name of the script the testplan is running

        metadata attributes:

            start_time: (datetime) time testplan started running

            end_time: (datetime) time testplan ended running

            combos_total: (int) number of combos in testplan

            success: (int) number of combos with success status
```

failure: (int) number of combos with failure status

attention: (int) number of combos with attention status

skipped: (int) number of combos with skipped status

error: (int) number of combos with error status

purged: (int) number of combos with purged status

running: (int) number of combos with running status

pending: (int) number of combos with pending status

p_notes: (str) notes on the testplan

keywords: [(str)] keywords associated with the namespace

list_vlans(self, with_header=False, filter_by=None, sort_by=None, display_num=None, cut_to=None, select=None) from tyworkflow.resource_manager.client_util.DB

Lists vlans. See list_attr for detailed parameter descriptions.

Valid fields to select/filter are:

formal attributes:

id: (int) ID of this thing

metadata attributes:

name: (str) name of the vlan; where applicable, should match the name of the network to which the computer's network interface is connected (e.g. the network name on an ESXi host)

ip_min: (str) minimum IP address bounding a pool of IP addresses from which pick a unique one when needed (e.g. cloning)

ip_max: (str) maximum IP address bounding a pool of IP addresses from which pick a unique one when needed (e.g. cloning)

mac_min: (str) minimum MAC address bounding a pool of MAC addresses from which pick a unique one when needed (e.g. cloning)

mac_max: (str) maximum MAC address bounding a pool of MAC addresses from which pick a unique one when needed (e.g. cloning)

list_vm_hosts(self, with_header=False, filter_by=None, sort_by=None, display_num=None, cut_to=None, select=None) from tyworkflow.resource_manager.client_util.DB

```
    Lists VM hosts. See list_attr for detailed parameter descriptions. Valid

    fields to select/filter are:

        formal attributes:

            id: (int) ID of this thing

        metadata attributes:

            host: (str) hostname/IP address to connect to the VM host

            type: (str) type of VM host (e.g. esxi, vbox, etc)

            user: (str) username used to log in to the VM host

            passwd: (str) password used to log in to the VM host

            max_in_use: (int) maximum number of VMs in use concurrently for
automated tests on this host; set <= 0 or NULL for no limit



migrate_db(self, *args) from tyworkflow.resource_manager.client_util.DB

    Migrate tyrant database from previous version.

    The argument list must include the magic string "+really-do-it" to help
prevent

    accidental execution of this command, or optionally, you will be prompted
for

    confirmation prior to command execution.



post_clone_status(self, operation, c_name, snapshot, reserve_name) from
tyworkflow.resource_manager.client_util.DB



purge_clone_status(self, clone_status_id) from
tyworkflow.resource_manager.client_util.DB



purge_computers(self, *args) from tyworkflow.resource_manager.client_util.DB

    Purge the tyrant computer,  snapshot, and computer attr table entries.

    The argument list must include the magic string "+really-do-it" to help
prevent
```

```
    accidental execution of this command, or optionally, you will be prompted
for

    confirmation prior to command execution.


purge_metrics(self, *args) from tyworkflow.resource_manager.client_util.DB

    Purge the tyrant test result table entries.

    The argument list must include the magic string "+really-do-it" to help
prevent

    accidental execution of this command, or optionally, you will be prompted
for

    confirmation prior to command execution.


purge_recipes(self, *args) from tyworkflow.resource_manager.client_util.DB

    Purge the tyrant recipe,  snapshot, and recipe attr table entries.

    The argument list must include the magic string "+really-do-it" to help
prevent

    accidental execution of this command, or optionally, you will be prompted
for

    confirmation prior to command execution.


purge_resources(self, *args) from tyworkflow.resource_manager.client_util.DB

    Purge the tyrant resource table entries (computer, recipe, snapshot).

    The argument list must include the magic string "+really-do-it" to help
prevent

    accidental execution of this command, or optionally, you will be prompted
for

    confirmation prior to command execution.


purge_snapshots(self, *args) from tyworkflow.resource_manager.client_util.DB

    Purge the tyrant snapshot and snapshot attr table entries.

    The argument list must include the magic string "+really-do-it" to help
prevent
```

```
    accidental execution of this command, or optionally, you will be prompted
for

    confirmation prior to command execution.


purge_tests(self, *args) from tyworkflow.resource_manager.client_util.DB

    Purge the tyrant test result table entries.

    The argument list must include the magic string "+really-do-it" to help
prevent

    accidental execution of this command, or optionally, you will be prompted
for

    confirmation prior to command execution.


query(self, query, *params) from tyworkflow.resource_manager.client_util.DB

    Runs the given query, with the given parameters. If the query is

    successful, the results of the query are returned (as an iterable of

    rows, where each row is itself an iterable). If the query fails, an

    an exception (StandardError of some child of it) is raised.


    Parameters:

        str query: the SQL query to execute. If this query contains

            database-backend-specific parameter substitution strings (e.g.

            "%s" for MySQL or "?" for SQLite), then the elements of the

            "params" parameter will be substituted in for them using the

            underlying database engine's parameter substitution.

        [obj] params: list of parameters to the query, which will be

            substituted into the query using the DB engine.


    Returns: an iterable of rows of results from the query. For some

    queries, there are no results (e.g. DROP TABLE), in which case the
```

```
    returned value is an empty list.


    Raises: Generally, a DB-API 2.0-compliant subclass of python's
    StandardError. In the case that the database engine detects the
    error was due to insufficient privileges, then
    .engines.PrivilegeError is raised.


release_clone_lock(self) from tyworkflow.resource_manager.client_util.DB


reserve_asset(self, reserve_name='', **attrs) from
tyworkflow.resource_manager.client_util.DB
    Reserves assets (computers).


    Parameters:
        str reserve_name: name to store on the computer record to identify
who's
            reserved it
        **attrs: keyword args specifying values to match on computer fields.
            Keys are the names of valid computer fields (see list_computers),
            values are what those fields must equal. Only matching computers
            will be reserved.


    Examples:
        reserve a computer with a specific id
            bin/db_admin reserve_asset my_name id=104
        reserve all computers in a specific pool
            bin/db_admin reserve_asset my_name pool=pool001
        reserve all computers not currently reserved
            bin/db_admin reserve_asset my_name testing_use=Y
```

```
reset_clone_status(self) from tyworkflow.resource_manager.client_util.DB



set_clone_status_name(self, status_id, name) from
tyworkflow.resource_manager.client_util.DB

    Sets the clone name for the clone status entry with the given name.



set_computer_attr(self, attr_table, attr_name, **attrs) from
tyworkflow.resource_manager.client_util.DB

    Sets the given formal attribute for a computer.



    Parameters:

        str attr_table: name of the formal attribute being set

        str attr_name: the value of the attribute being set

        {str, str} **attrs: Keyword arguments specifying formal or metadata

            attributes and their values. If specified, then only entries with

            attributes matching the values given here will be updated.
Otherwise,

            all entries are updated.



    Valid keywords for **attrs are the formal attribute names listed in the
help for

    list_computers, or ANY of the extended hwtype-specific attributes also
listed

    there.



set_computer_flag(self, flag_name, flag, **attrs) from
tyworkflow.resource_manager.client_util.DB

    Sets the given flag (metadata attribute) for a computer.



    Parameters:
```

```
        str flag_name: name of the flag to set

        str flag: Value to set for the flag.

        {str, str} **attrs: Same as set_computer_attr.


    Valid keywords for **attrs are those metadata attributes listed in

    list_computer's help which are flags (e.g. take 'Y', 'N', or 'R'

    values).


set_dbuser_password(self, name, password=None) from
tyworkflow.resource_manager.client_util.DB

    Sets the password of the specified database user.


    Parameters:

        str name: name of the user whose password should be set

        str password: the new password to set


    Returns: True on success

    Raises: StandardError on error


set_dbuser_privlev(self, name, level) from
tyworkflow.resource_manager.client_util.DB

    Sets the user's privilege level to the given privilege level.


    WARNING: This command should only be used with MySQL user accounts which

    were created with db_admin's add_dbuser command, as it revokes all

    privileges for the specified user before adding back the specific

    privileges for the specified level.
```

```
    Parameters:

        str name: name of the user

        str level: level identifier, one of the following:

            view: read-only access to the database; allows one to view test
results but not run tests

            test: allows one to execute tests and perform other operations a
tester might want to do

            admin: gives full control of the database



    Returns: True on success

    Raises: StandardError on error



set_pdu_attr(self, attr_name, attr_val, **attrs) from
tyworkflow.resource_manager.client_util.DB

    Sets the given field (attr_name) to the given value (attr_val) for

    all PDU entries matching the given filters (attrs; a dict keyed by

    field name, whose values determine attributes an entry must have to be

    modified by this method). Valid fields to set are those listed in the

    help for list_pdus.



set_recipe_attr(self, attr_table, attr_name, **attrs) from
tyworkflow.resource_manager.client_util.DB

    Sets the given attribute for a recipe. See set_computer_attr for

    detailed parameter descriptions. Valid **attrs values are the formal

    attribute names listed in list_recipes's help.



set_resource_flag(self, flag_name, flag, **attrs) from
tyworkflow.resource_manager.client_util.DB

    Sets the given metadata attribute for a computer/snapshot.
```

```
    Parameters:

        str flag_name: name of the flag to set

        str flag: Value to set for the flag.

        {str, str} **attrs: Keyword arguments specifying formal or metadata

            attributes and their values. If specified, then only entries with

            attributes matching the values given here will be updated.
Otherwise,

            all entries are updated.


    Valid keywords for **attrs are the metadata attribute names listed in

    list_resource's help which take flag values (e.g. 'Y', 'N', 'R').


    Examples:

        clear fubar flag on all resources in pool1:

            bin/db_admin set_resource_flag testing_fubar 'N' pool=pool1

        reserve computer with id 100:

            bin/db_admin set_resource_flag testing_use 'N' computer_id=100


set_snapshot_flag(self, flag_name, flag, **attrs) from
tyworkflow.resource_manager.client_util.DB

    Sets the given flag (metadata attribute) for a snapshot. See

    set_computer_flag for detailed parameter descriptions. Valid keywords

    for **attrs are the metadata attributes listed in list_snapshot's help

    which take flag values (e.g. 'Y' or 'N').


set_testcase_results(self, testcase_id, result_code=None, result=None,
change_comment=None) from tyworkflow.resource_manager.client_util.DB

    Sets results and/or result code for a testcase, given its id.

    Arguments:
```

```
        testcase_id : ID for testcase to change

        result_code : if this parameter is specified, will set specified value
for testcase

        result : if this parameter is specified, will set specified value for
testcase

        change_comment: if this parameter is specified, will set specified
value for testcase



    The first time result code and/or result is changes, the original result
code and results are stored.

    When results change, the current change timestamp is set.



set_value = __set_value(self, table_name, ATTRS, ATTR_OTHERS, value_name,
new_value, use_literal=False, **attrs) from
tyworkflow.resource_manager.client_util.DB

    Sets fields in a table to values.



    Parameters:

        str table_name: name of the table the field exists in

        str[] ATTRS: names of format attributes for entries in the table

        (str, str)[] ATTR_OTHERS: unused

        str value_name: name of the field to set

        str new_value: the new value to set

        bool use_literal: if True, then the given value (stringified if it's

            not already set) will be set exactly; if False, the given value

            will be passed through normal DBAPI parameter substitution; set

            use_literal True if you want to do things like set a field to

            NULL or CURRENT_TIMESTAMP

        See set_computer_attr for a description of **attrs.
```

```
set_vlan_attr(self, attr_name, attr_val, **attrs) from
tyworkflow.resource_manager.client_util.DB

    Sets the given field (attr_name) to the given value (attr_val) for
    all vlan entries matching the given filters (given in kwargs) whose
    values determine attributes an entry must have to be modified by this
    method). Valid fields to set are those listed in the help for
    list_vlans.


set_vm_host_attr(self, attr_name, attr_val, **attrs) from
tyworkflow.resource_manager.client_util.DB

    Sets the given field (attr_name) to the given value (attr_val) for
    all VM host entries matching the given filters (attrs; a dict keyed by
    field name, whose values determine attributes an entry must have to be
    modified by this method). Valid fields to set are those listed in the
    help for list_vm_hosts.


table_exists(self, table_name) from tyworkflow.resource_manager.client_util.DB

    Returns '1' if the database table exists, otherwise '0'


unreserve_asset(self, **attrs) from tyworkflow.resource_manager.client_util.DB

    Unreserves assets, making them available for testing.  **attrs is the
    same as for reserve_asset.


    Examples:
        unreserve all computers reserved with a specific name
            bin/db_admin unreserve_asset reserve_name=my_name
        unreserve all reserved assets
            bin/db_admin unreserve_asset testing_use=N
        unconditionally unreserve everything
```

```
            bin/db_admin unreserve_asset
```

7.1.3 overmind_admin

```
Usage: client.py [option]* <cmd> [<arg>]*


Options:

  -h           : Print help.

  -v           : Enable verbose mode. (logging.DEBUG)

  -q           : Disable verbose mode. (logging.WARN)

  -t <timeout> : Set command timeout value.

  -s <server>  : Server to connect to.

  -p <port>    : Port to connect to.


Methods defined here:


purge_plan(self, nid='', pid='', tid='') from
tyworkflow.overmind.commands.Commands

    Purge the plan, match on attributes (all integers):

        n(amespace)id

        p(lan)id

        t(est)id


service_echo(self, *args, **kargs) from
tyworkflow.overmind.commands.Commands

    Return arguments.


service_get_children(self) from tyworkflow.overmind.commands.Commands
```

```
    Retrieves the current maximum number of child processes in the
service.


service_get_popthreads(self) from
tyworkflow.overmind.commands.Commands

    Retrieves the current integer maximum number of threads running

    post-ops for tests.


service_log_level(self, log_level, *logs) from
tyworkflow.overmind.commands.Commands

    Call setLogLevel on the root logger and any other named loggers


service_ping(self) from tyworkflow.overmind.commands.Commands

    Return True.


service_set_children(self, max_children) from
tyworkflow.overmind.commands.Commands

    Set the integer maximum number of child processes in the service.


service_set_popthreads(self, max_popthreads) from
tyworkflow.overmind.commands.Commands

    Sets the integer maximum number of threads running post-ops for

    tests.


service_shutdown(self) from tyworkflow.overmind.commands.Commands

    Shuts down the Overmind server, but refuses to run if there are

    post-op threads in progress (since they cannot be safely stopped).


service_timestamp(self) from tyworkflow.overmind.commands.Commands
```

```
    Return service start time.


service_uptime(self) from tyworkflow.overmind.commands.Commands

    Return service uptime.


submit_plan(self, planmod, **plan_opts) from
tyworkflow.overmind.commands.Commands

    Submit the plan specified in the string planmod.
    Keyword arguments override plan attributes of the same name.
```

7.1.4 overmind

```
Usage: overmind [<option>]*


Options:

  -h           : Print help.

  -c <rc_file> : Use rc_file as an alternate config file.

  -l <host>    : Listen for connection on host ip.

  -p <port>    : Listen for connections on port.

  -o <dir>     : Output directory to use.

  -f           : Run overmind in the foreground (Default is as a
daemon).

  -b           : Run overmind in the background.

  -v           : Turn on verbose diagnostic output.

  -q           : Turn off verbose diagnostic output.

  -x           : Require at least one argument to start service.
```

7.1.5 reaper_admin

```
Usage: client.py [option]* <cmd> [<arg>]*


Options:
```

```
  -h           : Print help.

  -v           : Enable verbose mode. (logging.DEBUG)

  -q           : Disable verbose mode. (logging.WARN)

  -t <timeout> : Set command timeout value.

  -s <server>  : Server to connect to.

  -p <port>    : Port to connect to.


Methods defined here:


revert_host(self, **kwargs) from tyworkflow.reaper.commands.Commands

    Revert the computer designated by the given fields. Valid kwargs
are

    the fields that make up a resource entity (e.g. from db_admin

    list_resources).


service_echo(self, *args, **kargs) from
tyworkflow.reaper.commands.Commands

    Return arguments.


service_log_level(self, log_level, *logs) from
tyworkflow.reaper.commands.Commands

    Call setLogLevel on the root logger and any other named loggers


service_ping(self) from tyworkflow.reaper.commands.Commands

    Return True.


service_set_children(self, max_children) from
tyworkflow.reaper.commands.Commands
```

```
    Set the maximum number of concurrent reverts allowed.

service_shutdown(self) from tyworkflow.reaper.commands.Commands
    Shut the server down.

service_timestamp(self) from tyworkflow.reaper.commands.Commands
    Return service start time.

service_uptime(self) from tyworkflow.reaper.commands.Commands
    Return service uptime.
```

7.1.6 reaper

```
Usage: reaper [<option>]*

Options:
  -h           : Print help.
  -c <rc_file> : Use rc_file as an alternate config file.
  -p <port>    : Listen for connections on port.
  -m <num>     : Maximum number of concurrent reverts.
  -f           : Run reaper in the foreground (default is as a daemon).
  -b           : Run reaper in the background.
  -s <args..>  : Start in single pass mode to revert the host described by a
                 list of key/value pairs giving information about the host
                 to the given snapshot. The snapshot comes first, then the
                 key/value pairs (-s <snapshot> <key=val ..>).
  -v           : Turn on verbose diagnostic output.
  -q           : Turn off verbose diagnostic output.
  -x           : Require at least one argument to start service.
```

```
This tool is intended primarly for use with/by the reaper server. If you
wish to revert an asset to a stored state, use the comp_admin tool in
tybase.

Methods defined here:

revert_host(self, **kwargs) from tyworkflow.reaper.commands.Commands
    Revert the computer designated by the given fields. Valid kwargs are
    the fields that make up a resource entity (e.g. from db_admin
    list_resources).

service_echo(self, *args, **kargs) from tyworkflow.reaper.commands.Commands
    Return arguments.

service_log_level(self, log_level, *logs) from
tyworkflow.reaper.commands.Commands
    Call setLogLevel on the root logger and any other named loggers

service_ping(self) from tyworkflow.reaper.commands.Commands
    Return True.

service_set_children(self, max_children) from
tyworkflow.reaper.commands.Commands
    Set the maximum number of concurrent reverts allowed.

service_shutdown(self) from tyworkflow.reaper.commands.Commands
    Shut the server down.
```

```
service_timestamp(self) from tyworkflow.reaper.commands.Commands

    Return service start time.


service_uptime(self) from tyworkflow.reaper.commands.Commands

    Return service uptime.
```

7.1.7 plunger_admin

```
Usage: client.py [option]* <cmd> [<arg>]*

Options:
  -h           : Print help.
  -v           : Enable verbose mode. (logging.DEBUG)
  -q           : Disable verbose mode. (logging.WARN)
  -t <timeout> : Set command timeout value.
  -s <server>  : Server to connect to.
  -p <port>    : Port to connect to.

Methods defined here:

resync_db_time(self) from tyworkflow.plunger.commands.Commands
    Syncronizes service time with DB time

service_echo(self, *args, **kargs) from
tyworkflow.plunger.commands.Commands
    Return arguments.

service_log_level(self, log_level, *logs) from
tyworkflow.plunger.commands.Commands
    Call setLogLevel on the root logger and any other named loggers

service_ping(self) from tyworkflow.plunger.commands.Commands
    Return True.

service_shutdown(self) from tyworkflow.plunger.commands.Commands
    Shut the server down.

service_timestamp(self) from tyworkflow.plunger.commands.Commands
    Return service start time.

service_uptime(self) from tyworkflow.plunger.commands.Commands
    Return service uptime.

update_contention_age(self, age) from
tyworkflow.plunger.commands.Commands
    Changes contention age (for deleting contentions older than # of
days)
```

```
update_large_interval(self, interval) from
tyworkflow.plunger.commands.Commands
    Changes large interval hours (for rolling into large contentions
within interval)

update_reservation_age(self, age) from
tyworkflow.plunger.commands.Commands
    Changes reservation history age (for deleting reservation history
older than # of days)

update_small_interval(self, interval) from
tyworkflow.plunger.commands.Commands
    Changes small interval minutes (for rolling contentions within
interval)
```

7.1.8 plunger

```
Usage: plunger [<option>]*


Options:

  -h           : Print help.

  -c <rc_file> : Use rc_file as an alternate config file.

  -p <port>    : Listen for connections on port.

  -f           : Run plunger in the foreground (default is as a daemon).

  -b           : Run plunger in the background.

  -v           : Turn on verbose diagnostic output.

  -q           : Turn off verbose diagnostic output.

  -x           : Require at least one argument to start service.


This tool is intended primarily for use with/by the plunger server. If you

wish to perform other db cleanup functions, use the db_admin tool.
```

5 Tybase

7.1.9 palantir_admin

```
Usage: client.py [option]* <cmd> [<arg>]*
```

```
Options:

  -h           : Print help.

  -v           : Enable verbose mode. (logging.DEBUG)

  -q           : Disable verbose mode. (logging.WARN)

  -t <timeout> : Set command timeout value.

  -s <server>  : Server to connect to.

  -p <port>    : Port to connect to.


Methods defined here:


clone(self, reuse_session=False) from tybase.palantir.client.Client

    Creates and opens a new Client object to (self.host,self.port).

    + reuse_session flag causes the session to be reused by the copy.

    + WARNING: client.close() will close the session on the server for all
copies.


createEmissary(self, domain=None, username=None, password=None) from
tybase.palantir.client.Client

    Returns a palantir client object connected to the user instance of

    Palantir on the host. If the user instance has not been set up, setup is

    attempted.


    Parameters:

        str domain: Domain of the user we want to run as. If not set, then the

            domain component is ignored entirely, both for logging in a new
user

            and for checking whether the currently-logged-in user is the right

            one.

        str username: Username of the user we want to run as. If that user is
```

```
            not currently logged-in, the currently-logged-in user will be
logged
            out and the given one logged in. If not set and a user is
currently
            logged in, that user will be used, otherwise (if no user is logged
            in and no user is given, an exception will be raised). Also, if
not
            set, the domain component is ignored as well.
        str password: Password to use for logging in as the given user.


execcmd(self, *args, **kargs) from tybase.palantir.client.Client
    Runs a command remotely, waiting for it complete.


execfunc(self, path, *args, **kargs) from tybase.palantir.commands.Commands
    Return path(*args, **kargs)


    An execfunc user has implicit access to these variables:
      self            --> palantir.commands.Commands( ...
support.netcom.ServerCommands , threading.Thread )
      self.server     --> palantir.server.Server( support.netcom.Server )
      self.command_id --> (NEW) this command_id currently being processed


    An execfunc user has implicit access to these functions:
      self.set_session_variable()
      self.get_session_variable()
      self.ANY_METHOD_IN_COMMANDS_SUBCLASS -- e.g. execfunc from
palantir/commands/Commands


execute(self, opts, args) from tybase.palantir.commands.Commands
    Runs a command (e.g. a new process) on the remote side.
```

The elements of the command line are given as positional arguments in *args. Specify your command-line already tokenized, as you would when using python's subprocess module (which is what's used on the remote side).

opts and **kargs serve the same purpose. They let you specify keyword arguments to affect how the command is run on the remote side. Valid keywords and their meaning are as follows:

    bool wait: whether to wait for the command to complete before
        execute returns; defaults True
    str cwd: current working directory for running the command;
        defaults to the current working directory at the time of
        invocation
    str stdin: path to a file on the remote side from which to read
        data to the process's standard in
    str stdout: path to a file on the remote side to which to write
        the process's standard out
    str stderr: path to a file on the remote side to which to write
        the process's standard error
    bool detach: whether to detach the remote process from the palantir
        server which invoked it; defaults True
    bool shell: if True, invoke the command in the shell; use this if
        you need to use shell pipelining or for some other reason have
        to specify the command as one big string; defaults False. If
        setting shell True, you almost certainly want to specify your
        command as one string rather than pre-tokenized. If you set
        shell=True and specify a list of commands, behavior will differ

```
            based on the target operating system; read the python subprocess
            module documentation for more info.
    The values in kargs are merged into opts, you can specify these options
    as either elements of the opts dict, or as keyword arguments to execute.


externfunc(self, path, *args, **kargs) from tybase.palantir.client.Client
    Runs a blob of python code on the remote side.


fappend(self, rfile, data) from tybase.palantir.client.Client
    Appends data to remote file


fhash(self, fname, method='md5', offset=0, nbytes=0) from
tybase.palantir.commands.Commands
    Return hash via methods [md5|crc32]


flength(self, rfile) from tybase.palantir.client.Client
    Returns the size of the remote file


fread(self, fname, offset=0, nbytes=0) from tybase.palantir.commands.Commands
    Return file contents.


fwrite(self, fname, fdata, offset=0) from tybase.palantir.commands.Commands
    Write file contents.


get(self, rfile, lfile=None, mode=None, force=True, max_bytes=None) from
tybase.palantir.client.Client
    Get the remote file and store its contents locally.


    rfile    : The remote file to get.
```

```
    lfile    : The local file that will be written.

    mode     : File permissions.

    force    : Always write file (do not test).

    max_bytes: Number of bytes to read at once (chunk size for a read loop)


    Return number of bytes read.


get_os_arch(self) from tybase.palantir.client.Client

    Returns the standardized CPU architecture name (e.g. x86, x86_64, ia32,
etc) of the OS.


get_os_family(self) from tybase.palantir.client.Client

    Returns the standardized OS family name (e.g. linux, windows).


get_platform(self) from tybase.palantir.client.Client

    Wrapper to remote execution of sys.platform()


host_ping(self, tries=1, delay=5) from tybase.palantir.client.Client

    ICMP ping the remote host


mirrorfunc(self, *args, **kargs) from tybase.palantir.client.Client

    Runs a function on the remote side and gives back a proxy to the

    remote return value, allowing you to work with e.g. imported modules,

    open file handles, and other objects that can't be pickled and sent

    across the connection normally.


mkdir(self, name) from tybase.palantir.client.Client

    Make a remote directory
```

```
ostype(self) from tybase.palantir.client.Client

    Returns a string indicating the OS that the server is running.


path_exists(self, path) from tybase.palantir.client.Client

    Returns True if the given filesystem path exists on the remote side.


pathsep(self) from tybase.palantir.client.Client

    Wrapper to remote execution of os.pathsep()


put(self, lfile, rfile=None, mode=None, force=True, max_bytes=None,
remote_check=True) from tybase.palantir.client.Client

    Test and put the local file to remote file.


    lfile    : The local file to put.

    rfile    : The remote file that will be written.

    mode     : File permissions.

    force    : Always write file (do not test).

    max_bytes: Number of bytes to write at once (chunk size for the upload
loop)


    Return number of bytes written.


remotefunc(self, func, *args, **kargs) from tybase.palantir.client.Client

    Given a function object, runs it remotely and gives back the return

    value.


rget(self, src_path, tgt_path=None, **kargs) from
tybase.palantir.client.Client
```

```
    Get files/directories at src_path to tgt_path (using rysnc module)


rmdir(self, name) from tybase.palantir.client.Client

    Delete a remote directory


rmfile(self, name) from tybase.palantir.client.Client

    Delete a remote file


rmtree(self, name) from tybase.palantir.client.Client

    Delete a remote directory tree


rput(self, src_path, tgt_path=None, **kargs) from
tybase.palantir.client.Client

    Put files/directories at src_path to tgt_path (using rysnc module)


sep(self) from tybase.palantir.client.Client

    Wrapper to remote execution of os.sep()


service_echo(self, *args, **kargs) from tybase.palantir.commands.Commands

    Return arguments.


service_log_level(self, log_level, *logs) from
tybase.palantir.commands.Commands

    Call setLogLevel on the root logger and any other named loggers


service_ping(self) from tybase.palantir.commands.Commands

    Return True.


service_secure_cert_fname(self) from tybase.palantir.commands.Commands
```

```
    Incoming format:

    Returns the string filename for the trusted SSL
    certificate file.

service_shutdown(self) from tybase.palantir.commands.Commands
    Shut the server down.

service_timestamp(self) from tybase.palantir.commands.Commands
    Return service start time.

service_uptime(self) from tybase.palantir.commands.Commands
    Return service uptime.

service_version(self) from tybase.palantir.commands.Commands
    Return server version.

shutdown(self) from tybase.palantir.commands.Commands
    Shuts down the palantir server.

spawn(self, *args, **kargs) from tybase.palantir.client.Client
    Non-blocking execution of remote commands

sys_executable(self) from tybase.palantir.client.Client
    Wrapper to remote execution of sys.executable()

system(self, *args, **kargs) from tybase.palantir.client.Client
    Runs a command, remotely, through a shell, waiting for it to
```

```
    complete.  The command SHOULD be specified as a single string, NOT
    tokenized.  The command MAY be specified as multiple tokens, in which
    case the tokens will be joined on a space.  Therefore, if specified as
    tokens, each token must be individually quoted properly in order for
    system to work.
```

7.1.10 palantir

```
Usage: palantir [option]*


Options:


-h            : Print help.
-c <rc_file>  : Use rc_file as an alternate config file.
-L <log_file> : Write log messages to log_file
-t <cert_file>: Set trusted SSL/TLS certificate file
-l <host>     : Listen for connection on host ip.
-p <port>     : Listen for connections on port.
-f            : Run palantir in the foreground.
-b            : Run palantir in the background.
-v            : Verbose output on.
-q            : Verbose output off.
-x            : Require at least one argument to start service.
-K            : Creates and registers a new SSL cert for local
connections.
-S            : Use secure transport mode
```

7.1.11 plundermine

```
Usage: plundermine [<option>]* <script_name>[,<script_name>]
[host_slot]* [-- [arg_slot]*]

       host_slot  =  host[,host]*

       host       =  [ip|name|file:filename|-]

       arg_slot   =  arg[,arg]*


Options:

  -h           : Print help.

  -v           : Enable verbose mode (script).

  -V           : Enable verbose mode (script + palantir).

  -q           : Disable verbose mode.

  -l <dir>     : Directory log files are located in.

  -t <timeout> : Timeout before exiting.

  -p <port>    : Port to connect to.

  -m <mode>    : Recursively set mode on output directory.

  -c <num>     : Maximum number of concurrent undermines.

  -d           : Debug mode (print undermines to execute).

  -S           : Use secure transport mode.

  -N           : Use non-secure transport mode.
```

7.1.12 undermine

```
Command-line interface to the undermine system. Allows running
leafnodes

with various settings, and also the interactive undermine shell.
```

```
Usage: undermine [option]* <script_name> [host]* [-- [args]*
[kwargs]*]



Options:

  -h             : Print help.

  -v             : Enable verbose mode (script).

  -V             : Enable verbose mode (script + palantir).

  -q             : Disable verbose mode.

  -l <dir>       : Set output directory (directory log files and
output

                   files are written to). (string)

  -t <timeout>   : Timeout before exiting. (int)

  -p <port>      : Port to connect to. (int)

  -m <mode>      : Recursively set mode on output directory. (string
mode specification)

  -M             : test and set dirty mark on hosts

  -s <sessionId> : Enable interactive shell and set first sessionId

                   (if enabled, host, args, and kwargs are ignored)
(string?)

  -X             : Shut down the assets at the end of the tests

  -S --presetup [scriptArgs] : Setup pre-condition test script list
with arguments

  -C --postcleanup [scriptArgs] : Setup post-condition test script
list with arguments



host args are in the format

    host[?hwinfo]

where:

    -host is the IP address or hostname of the host to connect to
```

```
    -hwinfo is an optional dict of key/value pairs giving information
about

    the host. These are used by the HAL, and are exactly equivalent to
the

    "comp" argument used by comp_admin, with the addition that the IP

    address or hostname of the computer is used as the "ip" field if
hwinfo

    is not given. See "bin/comp_admin -h" for more information on how
to

    specify hwinfo.



presetup and postcleanup args are in the format

    [(script_name,hosts,args,kwargs)]

where:

    -hosts is a string list of at least one host slot in for format:

        ['hosts[n]','hosts[n-1]'] where n is the index of hosts of
parent script




args are specified in one of three forms, depending upon the
characters

preceding the value of the argument:

    @@: Indicates a raw string, e.g:

        @@some_val

        @@'C:\Documents and Settings\All Users'

    @: Indicates the value is an expression which will be evaluated,
e.g.:

        @True #(for a boolean)

        @100.5 #(for an float)
```

```
        @'"some string"' #(for a string, note how inner quoting is
required
            since it will be evaluated)
        @"['foo', True, 35]" #(for a list, explained further on)
    No preceding characters: Indicates a raw string, e.g.:
        some_val
        'some message string'


kwargs are specified in the form name=value, where name is the name of
the
keyword argument being set, and value is the value specified just as
with
args above, e.g.:
    keep=@False
    delay=@25
    failure_msg='something went wrong'
    test_args=@"['foo', 'bar', 'baz']" (for specifying a list)
    test_kwargs=@"{key1=val1, key2=val2}" (for specifying a dict)


Notes on quoting:
    There are two logical levels at which quotes may be required.
First, you
    need quotes around values which have spaces in them to get those
values
    into the argument parser intact.  For example, the following two
    commands will see different arguments:
        bin/undermine some.script host1 -- 'foo bar'
        bin/undermine some.script host1 -- foo bar
```

The first will see a single argument, 'foo bar'. The second will see

two arguments, 'foo' and 'bar'.

Second, in evaluated arguments, you may need inner quotes so that the

argument parser which evaluates your value knows how to behave. For

example, if you want to specify a string as an evaluated argument, the

command

```
bin/undermine some.script host1 -- @'some string'
```

will generate the error

```
SyntaxError: Syntax error (line 1)
p=LexToken(keyname,'string',1,5)
```

This is because the parser will effectively see that string as a line of

code to be parsed, not the literal string "some string". Instead, the

correct way to specify it is like so:

```
bin/undermine some.script host1 -- @"'some string'"
```

The outer set of double quotes gets the entire argument value into the

parser intact, the inner quotes lets the parser know to treat it as a

string literal.

In list context, the parser has some intelligence and does not need

string list items which have no spaces to be quoted. E.g., the following

two commands will parse successfully and have the same effect:

```
        bin/undermine some.script host1 -- @['foo', 'bar', True, 1.5,
'a b']

        bin/undermine some.script host1 -- @[foo, bar, True, 1.5, 'a
b']
```

In both cases, undermine sees the arguments as a string literal 'foo', a

string literal 'bar', the python boolean True, the float 1.5, and a

string literal 'a b'.

Raw strings and quotes: There is some unusual behavior with quotes in

raw strings, where the parser removes quotes in some cases. I will

illustrate this by examples. The commands

```
        bin/undermine some.script host1 -- @@['foo']

        bin/undermine some.script host1 -- ['foo']
```

which you might expect to be able to do if you wanted the string literal

"['foo']" will actually result in the argument

```
        [foo]
```

The parser has stripped the outer set of matching quotes. Instead, to

accomplish an argument of "['foo']", you would have to do one of the

following commands

```
        bin/undermine some.script host1 -- @@"['foo']"

        bin/undermine some.script host1 -- @@["'foo'"]

        bin/undermine some.script host1 -- ["'foo'"]

        bin/undermine some.script host1 -- "['foo']"

        bin/undermine some.script host1 -- [\'foo\']
```

Notice how in the first four cases, the parser is stripping off the

outer set of quotes. The fifth case illustrates that you can also escape the quotes to keep the parser from stripping them.

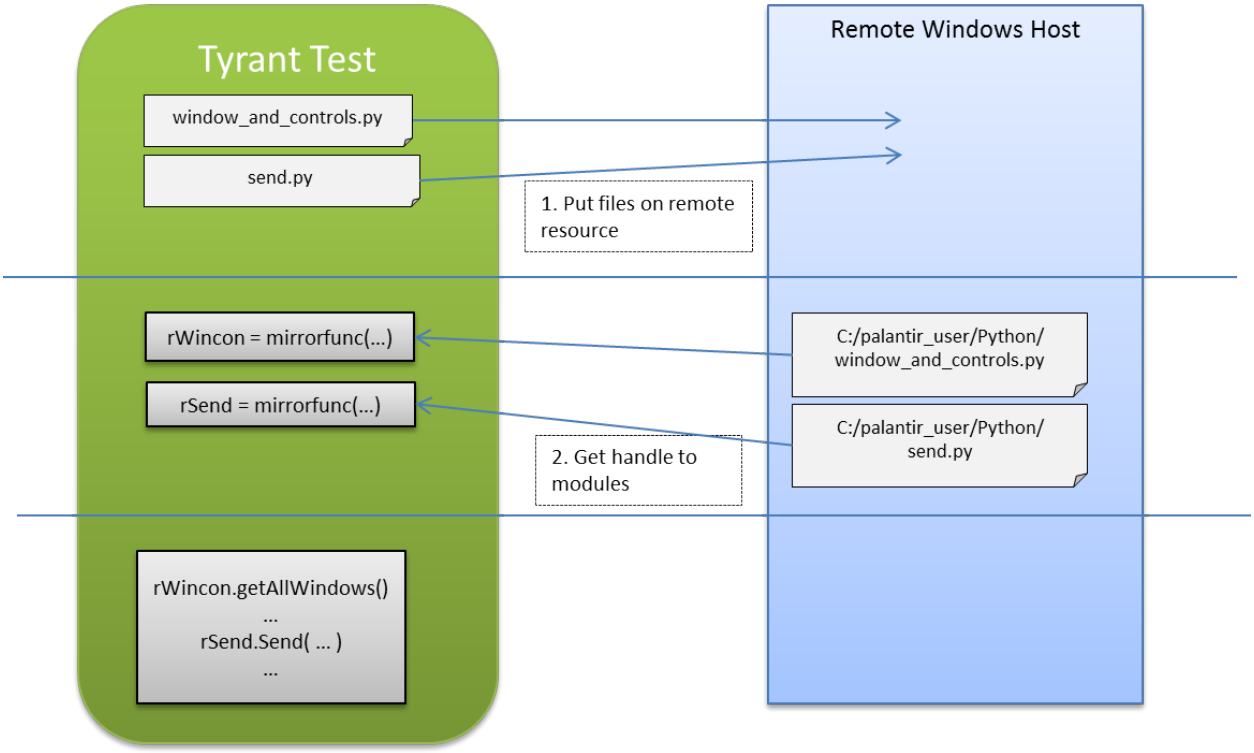
8 Appendix D – Window and Controls

Tyrant's window and controls libraries provide the ability to manipulate Windows graphical interface for automated testing. This document includes the requirements and example test scripts to allow window and controls testing in Tyrant as well as a brief description of API functions.

It is important to note that when writing a test function that utilizes the window and controls, there are three initial steps. This testing procedure is different than other Tyrant utility functions. The first step is to connect to the host as emissary. This allows the client to perform functions as a local user on the host, as opposed to performing functions as the system user.

The second initialization step is to put *send.py* and *window_and_controls.py* files into the Python directory on the host.

The third step is to call *rWinCon = mirrorfunc('__import__','window_and_controls')* and *rSend = mirrorfunc('__import__','send)*. The mirrorfunc() call establishes a handle to those functions from the remote host so that the test script can call functions from those modules. The sample test script in this document demonstrates these initialization steps.



8.1 *send.py*

This script provides functionality to perform key strokes and mouse movement and clicks on the remote host. The Send() command sends keystrokes to the host. When emulating a special key command (such

as pressing the Spacebar or “Enter” key, you must surround the command string with curly brackets. For example, to send the enter key, the function is called like so: `rSend.Send(“{ENTER}”)`

Below are the keys accepted by the Send() command:

```
'BACKSPACE': VK_BACK,
'BS': VK_BACK,
'TAB': VK_TAB,
'CLEAR': VK_CLEAR,
'RETURN': VK_RETURN,
"ENTER": VK_RETURN,
'SHIFT': VK_SHIFT,
'CONTROL': VK_CONTROL,
"SPACE": VK_SPACE,
"LEFT": VK_LEFT,
"UP": VK_UP,
"RIGHT": VK_RIGHT,
"DOWN": VK_DOWN,
"F1": VK_F1,
"F2": VK_F2,
"F3": VK_F3,
"F4": VK_F4,
"F5": VK_F5,
"F6": VK_F6,
"F7": VK_F7,
"F8": VK_F8,
"F9": VK_F9,
"F10": VK_F10,
"F11": VK_F11,
"F12": VK_F12,
"CAPSLOCK": VK_CAPITAL,
"SHIFTDOWN": VK_SHIFT,
"SHIFTUP": VK_SHIFT,
"CTRLDOWN": VK_CONTROL,
"CTRLUP": VK_CONTROL,
"ALT": VK_MENU,
"LALT": VK_LMENU,
"RALT": VK_RMENU,
"ALTDOWN": VK_MENU,
"ALTUP": VK_MENU,
"INSERT": VK_INSERT,
"DELETE": VK_DELETE,
"DEL": VK_DELETE,
"HOME": VK_HOME,
"END": VK_END,
"ESCAPE": VK_ESCAPE,
"ESC": VK_ESCAPE,
"BREAK": VK_CANCEL,
"PAUSE": VK_PAUSE,
"PLUS": VK_OEM_PLUS,
"MINUS": VK_OEM_MINUS,
```

```
    "PAGEDOWN": VK_NEXT,
    "PAGEUP": VK_PRIOR

''' keys which require a shift '''
    '!': ord('1'),
    '@': ord('2'),
    '#': ord('3'),
    '$': ord('4'),
    '%': ord('5'),
    '^': ord('6'),
    '&': ord('7'),
    '*': ord('8'),
    '(': ord('9'),
    ')': ord('0'),
    'A': ord('A'),
    'B': ord('B'),
    'C': ord('C'),
    'D': ord('D'),
    'E': ord('E'),
    'F': ord('F'),
    'G': ord('G'),
    'H': ord('H'),
    'I': ord('I'),
    'J': ord('J'),
    'K': ord('K'),
    'L': ord('L'),
    'M': ord('M'),
    'N': ord('N'),
    'O': ord('O'),
    'P': ord('P'),
    'Q': ord('Q'),
    'R': ord('R'),
    'S': ord('S'),
    'T': ord('T'),
    'U': ord('U'),
    'V': ord('V'),
    'W': ord('W'),
    'X': ord('X'),
    'Y': ord('Y'),
    'Z': ord('Z'),
    ':': VK_OEM_1,
    '?': VK_OEM_2,
    '~': VK_OEM_3,
    '{': VK_OEM_4,
    '|': VK_OEM_5,
    '}': VK_OEM_6,
    '"': VK_OEM_7,
    '+': VK_OEM_PLUS,
    '_': VK_OEM_MINUS,
    '<': VK_OEM_COMMA,
    '>': VK_OEM_PERIOD
```

8.2 *window_and_controls.py*

This script provides the functionality to perform actions on windows on the remote host.

Common windows functions include getting all windows, get a window by title, get a window by pid, move, maximize, minimize, and restore a window, activate and close a window. In to obtain a handle to a window, you must provide a valid Window title string. For example, when opening Notepad.exe for the first time, the window title that must be supplied to getWindowByTitle() is “Untitled – Notepad”.

8.3 Requirements

- Palantir instance on host which contains python win32api, win32gui, and win32 dependent libraries. (this already comes with Palantir)
- Tyrant setup with `tyutils` Repository.

8.4 Tyrant Window and Control API Functions

```
NAME
    window_and_controls

FILE
    c:/palantir_user/Python/window_and_controls.py

FUNCTIONS
       getChildWindows(hwnd)
           This function looks at the children of the passed in hwnd and returns the
           hwnd, title, and class of the windows. The EnumChildWindows function this
           uses is already recursive

       getAllWindows()

       getWindowByTitle(title)
           Gets a window by title and return the HWND
           If no window is found return 0

       getWindowByAppxTitle(title)
           Gets a window by title and return the HWND
           If no window is found return 0

       getWindowByClass(className, parentWin = None)
           Gets a window by class name and return the HWND
           Returns a list of matching windows
           Optionally specify a parent window to start the search from
           If no window is found return 0


       getWindowByPid(p)
           GetBy a window by PID and return the HWND
           If no window is found return 0


           Return only the first enabled window with text
           that matches the desired pid

       moveWindow(win, x, y, width=-1, height=-1)
```

```
    Move the window specified by the hwnd to a specific location

maximizeWindow(hwnd)

    Maximize the specified window


minimizeWindow(hwnd)
    Minimize the specified window


restoreWindow(hwnd, activate=True)
    Restore the specified window hwnd


showWindow(hwnd, sw=win32con.SW_SHOW)
    Show the specified window hwnd
    optional flag to min/max/restore


winWaitActive(win, maxWait=60, appxTitle=False)
    Wait for the window with the specified title or hwnd to become active
    Wait for a max of maxWait seconds - default is 60

    If the window becomes active we return immediately the hwnd
    If the timeout is reached before the window is activated return False


winWait(win, maxWait=60, appxTitle=False)
    Wait for the window with the specified title or hwnd to exist
    Wait for a max of maxWait seconds - default is 60

    If the window appears we return immediately the hwnd
    If the timeout is reached before the window is activated return False


winClose(win)
    Try and close the window specified


winWaitNotActive(win, maxWait=60)
    Wait for the window with the specified title or hwnd to become inactive
    Wait for a max of maxWait seconds - default is 60

    If the window becomes inactive we return True
    If the timeout is reached before the window is deactivated return False


winWaitClose(win, maxWait=60)
    Wait for the window with the specified title or hwnd to no longer exist
    Wait for a max of maxWait seconds - default is 60

    If the window goes away we return True
    If the timeout is reached before the window is deactivated return False


winActive(win)
    Check if the window with the specified title or hwnd is active
```

```
            If the window is active we return the hwnd
            If not we return 0


        winExists(win)
            Check if the window exists. Return True or False


        winActivate(win)
            Activate the window specified by the title or hwnd
            If successful this returns the hwnd
            If not this returns 0


        controlWait(win, ctlIdStr, instance, maxWait=60)
            Wait for the specified control to exist the amount of time
            specified (default 60 seconds)
            Return the control handle if it exists, 0 if we timeout


        getForegroundWindow()
            Get window at the forground


        winAttach(win, bAttach)
            bAttach to the window so that we can add things to the input queues

        getWindowRect(win)

        setCheckbox(hwndCtl, state=win32con.BST_CHECKED)
            Set or clear checkbox
            Do nothing if already in desired state
            Return new state, or 0 if request not possible


NAME
    send

FILE
    c:/palantir_user/Python/send.py

FUNCTIONS

        controlSend(win, ctlId, instance, text)
            Send input to the control that is a child of win given the control
            class and instance number
            The control class and instance number can be found with autoit's
            windowInfo program

            If you specify ctlId as an integer, then the code assumes it is a handle
            and it will not use the value entered for win or instance

        controlMsgSend(win, ctlClassStr, ctlInst, win32Msg, wParam, lParam)
            Attempt to send the control (known by ctlClassStr and ctlInst) that
            is a child of win

            EXAMPLE - called from a leadnode running on Linux machine interacting
            with a windows guest VM
            self.Rsend.controlMsgSend(self.mainHwnd, "MsoCommandBar", 1,
              self.Rwin32con.WM_CLOSE, 0, 0)
```

```
        MouseMove(x, y, speed=20)
            Move the mouse to the specified location at the given speed higher
            is faster


        Click(button="left", x=None, y=None, num=1, speed=20)
            Clicks the mouse button num times at the specified coordinates.
            If no coordinates are specified then it clicks the mouse in the
            current location.
            Speed is how fast the mouse moves to the location to click higher=faster


        ClickDrag(button, x1, y1, x2, y2, speed=20)
            Perform a mouse click drag operation
            Speed is how fast the mouse moves to the location to click higher=faster

        controlClick(win, ctlId, instance, numClicks, speed=20, offsetX=0, offsetY=0)
            Click the control that is a child of win given the control and
            instance number \n
            The control class and instance number can be found with autoit's
            windowInfo program

        getScreenMetrics()

        SetSendKeyDownDelay(val)


        SetSendKeyDelay(val)

        SetSendKeyDelayInterval(val)

        Send(keys, raw=0, ucode=0)

        windowSend(win, text)
            Send input to the given window

DATA
        SendKeyDownDelay  = 0.005
            5 ms default value
        SendKeyDelay      = 0.005
            5 ms default value
        SendKeyDelayInterval  = 0
            # +/- margin

        vkword = {
            'BACKSPACE': VK_BACK,
            'BS': VK_BACK,
            'TAB': VK_TAB,
            'CLEAR': VK_CLEAR,
            'RETURN': VK_RETURN,
            "ENTER": VK_RETURN,
            'SHIFT': VK_SHIFT,
            'CONTROL': VK_CONTROL,
            "SPACE": VK_SPACE,
            "LEFT": VK_LEFT,
            "UP": VK_UP,
            "RIGHT": VK_RIGHT,
            "DOWN": VK_DOWN,
            "F1": VK_F1,
            "F2": VK_F2,
            "F3": VK_F3,
            "F4": VK_F4,
```

```
    "F5": VK_F5,
    "F6": VK_F6,
    "F7": VK_F7,
    "F8": VK_F8,
    "F9": VK_F9,
    "F10": VK_F10,
    "F11": VK_F11,
    "F12": VK_F12,
    "CAPSLOCK": VK_CAPITAL,
    "SHIFTDOWN": VK_SHIFT,
    "SHIFTUP": VK_SHIFT,
    "CTRLDOWN": VK_CONTROL,
    "CTRLUP": VK_CONTROL,
    "ALT": VK_MENU,
    "LALT": VK_LMENU,
    "RALT": VK_RMENU,
    "ALTDOWN": VK_MENU,
    "ALTUP": VK_MENU,
    "INSERT": VK_INSERT,
    "DELETE": VK_DELETE,
    "DEL": VK_DELETE,
    "HOME": VK_HOME,
    "END": VK_END,
    "ESCAPE": VK_ESCAPE,
    "ESC": VK_ESCAPE,
    "BREAK": VK_CANCEL,
    "PAUSE": VK_PAUSE,
    "PLUS": VK_OEM_PLUS,
    "MINUS": VK_OEM_MINUS,
    "PAGEDOWN": VK_NEXT,
    "PAGEUP": VK_PRIOR
}

vkshift = {
    '!': ord('1'),
    '@': ord('2'),
    '#': ord('3'),
    '$': ord('4'),
    '%': ord('5'),
    '^': ord('6'),
    '&': ord('7'),
    '*': ord('8'),
    '(': ord('9'),
    ')': ord('0'),
    'A': ord('A'),
    'B': ord('B'),
    'C': ord('C'),
    'D': ord('D'),
    'E': ord('E'),
    'F': ord('F'),
    'G': ord('G'),
    'H': ord('H'),
    'I': ord('I'),
    'J': ord('J'),
    'K': ord('K'),
    'L': ord('L'),
    'M': ord('M'),
```

```
        'N': ord('N'),
        'O': ord('O'),
        'P': ord('P'),
        'Q': ord('Q'),
        'R': ord('R'),
        'S': ord('S'),
        'T': ord('T'),
        'U': ord('U'),
        'V': ord('V'),
        'W': ord('W'),
        'X': ord('X'),
        'Y': ord('Y'),
        'Z': ord('Z'),
        ':': VK_OEM_1,
        '?': VK_OEM_2,
        '~': VK_OEM_3,
        '{': VK_OEM_4,
        '|': VK_OEM_5,
        '}': VK_OEM_6,
        '"': VK_OEM_7,
        '+': VK_OEM_PLUS,
        '_': VK_OEM_MINUS,
        '<': VK_OEM_COMMA,
        '>': VK_OEM_PERIOD
    }
```

8.5 Example Tyrant Test Script

8.5.1 window_and_controls_test.py and window_and_controls_util.py

This test demonstrates basic window and controls functions. The window and controls test is dependent on *window_and_controls_util.py* to set up control scripts and establish a handle to those scripts on the remote host by calling the setup_wincon() function. The test creates emissary for the host, sets up control scripts, opens notepad using execcmd(), writing text in the notepad window, minimizes and restores the notepad window, moving the notepad window, saves the file using shortcut keys, and closes the notepad window.

NOTE: Before running the test for the first time, you must set **userName** and **userPwd** variables in the window_and_controls.py test script. This username and password should be credentials for a user on the testing machine. When running Overmind tests, the login credentials for every machine in a test must be that of the **userName** and **userPwd** set in the test script.

An example command line in Tybase to run this test with undermine:

```
bin/undermine tyutils.tests.window_and_controls_test 162.1.2.30
```

The code:

```
import time
import glob
import os

import tybase.undermine.meta.leafi as leafi
import tybase.undermine.main_script as main_script
from tyutils.window_and_controls_util import setup_wincon
```

```
@leafi.MainLeaf()
class WindowsAndControlsTest(main_script.Main_Script):
    userName = 'Administrator'
    userPwd = '#password#'
    saveFileName = 'c:\\test.txt'

    def run(self):
        result = self.SUCCESS
        msg = 'Finished.'

        if len(self.hosts) != 1:
            return(self.SKIPPED,'Must specify host.')


        #need to create emissary so that we're not running as 'system' user
        self.log.info("Creating Host Emissary")
        emhost = self.hosts[0].createEmissary(username=self.userName,password=self.userPwd)


        self.log.info("Setting up Windows and Controls")
        Rwincon,Rsend = setup_wincon(emhost)


        self.log.info("Opening Notepad")
        notepad_cmd = ['notepad.exe']
        rv = emhost.execcmd(notepad_cmd, shell=True, wait=False)
        time.sleep(3)


        notepad_app_title = 'Untitled - Notepad'
        self.log.info("Getting Notepad Window with title:",notepad_app_title)
        notepad_hwnd = Rwincon.getWindowByAppxTitle(notepad_app_title)


        self.log.info("Sending Text to Notepad Window")
        Rsend.windowSend(notepad_app_title,'Enter')

        self.log.info("Showing off by minimizing and restoring.")
        time.sleep(2)
        Rwincon.minimizeWindow(notepad_hwnd)
        time.sleep(2)

        Rwincon.restoreWindow(notepad_hwnd)

        rect = Rwincon.getWindowRect(notepad_hwnd)
        print rect
        self.log.info("Moving Notepad.")
        time.sleep(2)
        Rwincon.moveWindow(notepad_hwnd, rect[0]+40, rect[1]-40)

        self.log.info("Moving Back Notepad.")
        time.sleep(2)
        Rwincon.moveWindow(notepad_hwnd, rect[0], rect[1])

        if emhost.path_exists(self.saveFileName):
            self.log.warn(self.saveFileName,"Must have existed from previous test. Deleting.")
            #deleting file so that we don't have any additional unexpected window popups.
            emhost.rmfile(self.saveFileName)

        self.log.info("Saving File")
        Rsend.Send('{F10}')
        Rsend.Send('{DOWN}')
        Rsend.Send('A')
        time.sleep(2)
        Rsend.Send(self.saveFileName)
        Rsend.Send('{ENTER}')

        self.log.info("Closing Notepad")
        Rwincon.winClose(notepad_hwnd)


        return(result,msg)
```

8.5.2 Running Tests with Autoplan or Plan Files

To run a plan with window and controls tests, be sure to specify host slots that select windows hosts only.

In the example autoplan command line below, the host computer filter reaper value is the same as the host thumb drive pool value:

```
bin/autoplan solve tyutils.tests.window_and_controls_test -H family=windows samples=2
```

This should generate the following plan file:

```
---
from tyworkflow.support.planlang import *

auto_tc = TESTCASE(
    script = 'tyutils.tests.window_and_controls_test ',
    hostslots = [HOST(family='windows')],
    samples = 2,
    planname = 'auto_20140306115700'
)

EXECUTE(
    testcase = auto_tc
)
---
```

9 Appendix E - USB Testing

To perform tests using USBs, Tyrant includes scripts to be used with thumb drives attached to ESXi hosts. This document includes the requirements and setup to allow USB testing in Tyrant as well as a brief description of API functions.

Before testing with USBs, some initial setup is required. A usb thumb drive must be physically inserted into the ESXi Host Server. Then, on a virtual machine on the ESXi Host, manually add a USB controller and the USB device. This allows you to retrieve the port for that USB and the drive letter to create the contents of a usb blueprint file. A *usb_blueprint.rc* file must be created upon initial setup on each USB used for Tyrant testing. This USB drive must be plugged into the ESXi server at all times in order for Tyrant tests to be run against it.

After retrieving the USB port on ESXi, a resource for the USB thumb drive must be added to the Overmind database. This resource entry includes the ESXi reaper name and port specified in the IP address field. The Tyrant server must also include the *usb.rc* file.

A USB can only be tested with VMs on the same ESXi Host it is physically connected to. A USB thumb drive can only be used in one test at a time, in order to prevent resource usage collisions. When a tester is using a USB in an Undermine test, he or she must reserve that USB first in Overmind, and other testers should not use it in their own Undermine tests. If a thumb drive resource is not reserved, then Overmind can schedule tests which use that USB resource.

An ESXi server may have more than one USB slot. If so, more than one USB drive can be plugged into the ESXi server and used for testing.

A typical USB test involves calling the connect_usb() function, OR setup_usbs() function, to connect the USB to a computer, performing other actions, and then calling the disconnect_usb() function to disconnect the USB from the computer.

The connect_usb() function activates the USB drive on a host by essentially adding the USB device to the virtual machine, and then checking the host to determine which drive that USB device mapped to. The connect_usb() function returns the directory (which includes the drive letter in a Windows environment) to the new mapped drive. If the thumb drive is already in use by another virtual machine on the server, the default behavior of the function is to attempt to remove the USB device from the other virtual machine it is attached, and then add it again to the desired virtual machine. This default behavior of removing the thumb drive from another virtual machine can be overridden when the tester specifies “disconnect_other=False” when calling connect_usb(). This is demonstrated in *usb_test.py*.

To determine if the USB is attached ot another virtual machine, *usb_utils.py* calls another Tyrant internal script function which provides ESXi library functions, *vmesxilib.py*. If the tester wishes to call usb/esxi related library functions, please refer to *usb_utils.py*.

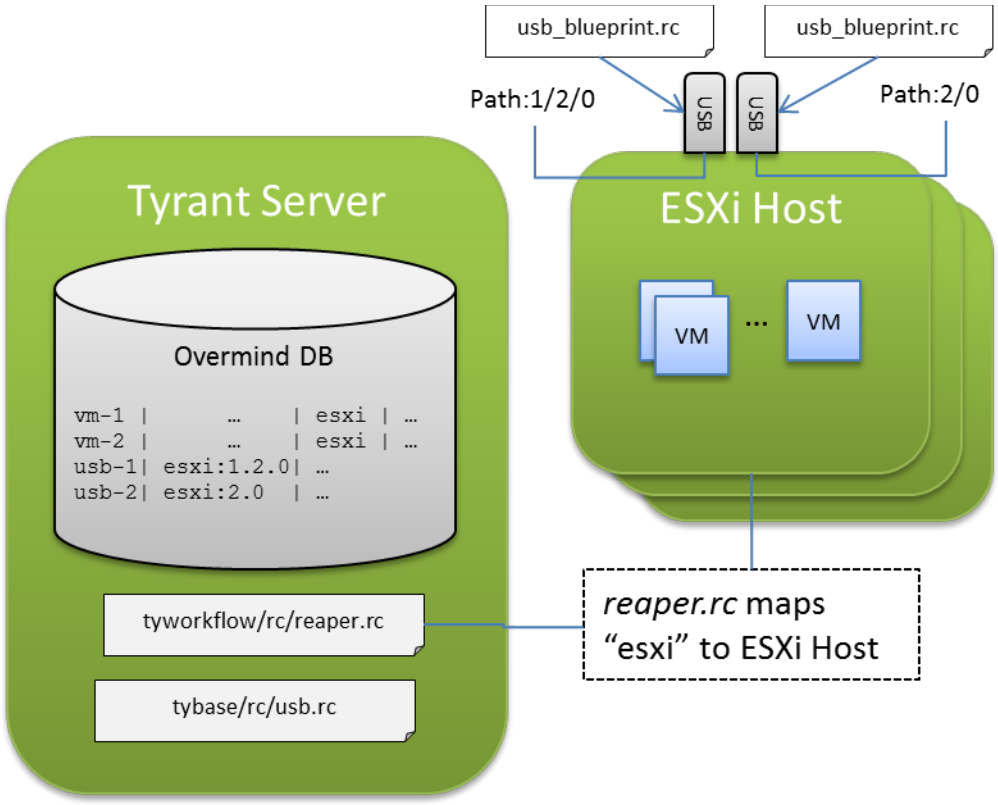
The disconnect_usb() function deactivates the USB drive on the host by removing the USB device from the virtual machine.

If connect_usb() function is successful, the function returns the directory for the mapped drive. This allows the tester to read and write files from the USB drive. In order to read and write files on the thumb drive, the tester can use Tyrant's host execute commands, or remote put file commands. A tester can also write a file directly on the USB drive. This is demonstrated in *usb_test.py*. For example, the following line in a test script to copy an existing file on a Windows host to the USB drive, "e:" may be:

```
host.execcmd('copy c:\\somefile.txt e:\\', shell=True)
```

The function, setup_usbs() allows the tester to wipe the contents of the USB to "clean" state. This function is ideally called in the beginning of a test. The setup_usbs() function relies on a *usb_blueprint.rc* file on the USB which provides a configuration of the files on the USB which are allowed to exist. These files listed would not be deleted from the USB. If a file exists on the USB that is not specified in the *usb_blueprint.rc* file, setup_usbs() will delete it.

A user can also verify the contents of the USB by calling provided functions such as check_usb_blueprint() and compare_usb_blueprint().



9.1 The usb.rc File

Tyrant testing requires the *usb.rc* file in Tybase's *rc* directory. Example content is as follows:

```
[dirty_file]
; Key can begin with [startswith, endswith, contains, equals]+'_'+*
startswith_1 = ~$7
endswith_2 = .bad
contains_3 = icky
equals_4 = ~$7icky.bad
```

The `dirty_file` section lists different file name pattern matching rules that will compare the contents of the USB drive (from reading the *usb_blueprint.rc* file on the USB) to that of the "dirty" listing in the usb.rc file.

This file name pattern match is performed when setup_usbs(), compare_usb_blueprint(), or check_usb_blueprint() functions are called. If a pattern match from one of the rules exists, an exception is thrown.

9.2 The usb_blueprint.rc File

In addition to connecting and disconnecting a usb to a virtual machine, Tyrant USB scripts allow testers to perform a "walk" of the USB as well as validate the state of the files on the thumb drive. In order to validate the state of the files on the USB, each USB thumb drive must have a USB blueprint file, *usb_blueprint.rc*, in its top level directory.

Example content is below:

```
[e:my_dir]
my_other_file.txt = 16

[e:]
my_file.txt = 512
usb_blueprint.rc = ignore_usb_bp
```

If the function create_usb_blueprint() is called, contents of the USB blueprint file are replaced with a mapping of the existing files.

When a user calls the setup_usbs() function, contents of the USB blueprint file is compared to the files and sizes on the USB thumb drive. Any files no recorded in the USB blueprint file are deleted and an exception is thrown.

All functions containing the name "blueprint", and the setup_usbs() function, handle a usb blueprint file.

9.3 Requirements

- USB Thumb Drive(s)
- ESXi Host Server (Must be using a commercial license (not a free version of ESXi))
- Hosts on that server managed by Overmind
- Tyrant setup with `tyutils` Repository.

NOTE: Tests can only be performed with USB drives and virtual machines connected to the same ESXi Host.

9.4 Setup

9.4.1 Setup USB on ESXi Server

1. Plug in USB to physical ESXi Server.
2. On a Windows VM in the ESXi Server:
    a. Select **Edit Settings** for VM.
    b. In **Virtual Machine Properties**, under **Hardware** tab, click **Add**.
    c. Select **USB Controller** device type, click **Next**.
    d. Choose all the default options by clicking next until **Finish**.
    e. In **Virtual Machine Properties**, click **OK** and wait for configuration changes to complete.
    f. Power on VM.
    g. Select **Edit Settings** for VM.
    h. In **Virtual Machine Properties**, under **Hardware** tab, click **Add**
    i. Select **USB Device** device type, click **Next**
    j. Select USB device listed in available devices, click **Next**.
    k. Continue to click next until **Finish**.
    l. Select the device being added, take note of the **USB Unique ID** and the port configuration after the "path" keyword.
    An example **USB Unique ID**:
    host: localhost path:1/2/0 version:2
    The port configuration is *1/2/0*.
    m. In **Virtual Machine Properties**, click **OK** and wait for configuration changes to complete.
    n. On the VM console, the USB device on the computer. Open the Computer and verify a new drive is created with the USB device. Note the drive letter. In some cases, this could be "**e:**".
    o. Manually create a file named *usb_blueprint.rc* on the USB drive. The contents of this file should be (replace the drive letter, e: with the drive letter of the usb drive. You can create multiple drive letter sections in the case that a thumb drive may map to different drives on different machines):

```
[e:]
usb_blueprint.rc = ignore_usb_bp
[f:]
usb_blueprint.rc = ignore_usb_bp
[/mnt/usb]
usb_blueprint.rc = ignore_usb_bp
```

    p. Gracefully remove the USB drive from the machine. To "unplug" the USB drive, go to **Edit Settings** for the VM, select the USB device added, and click **Remove**. Click **OK** to apply changes.

9.4.2 Setup USB Resource in Overmind

1. Create a *recipes_thumbs.csv* file with the thumb drive recipes.
   The recipe line format for a thumb drive is the following
   name,family,os,ossp,lang,arch,apps

   name – is the recipe name which combines all the recipe fields with '-'
   family – 'thumbdrive'
   os – the vendor
   ossp- the model
   lang -'td'
   arch - thumbdrive size
   apps – none, so leave blank

   Example line in *recipe_thumbs.csv* below:

   `thumbdrive-sandisk-cruzer-td-16gb,thumbdrive,sandisk,cruzer,td,16gb,`
   ---

2. Create a *computers_thumbs.csv* file with the thumb drive resource specifications
   The computer line format for a thumb drive is the following:
   `name,ip,mac,hwtype,pool,vlan,reaper`
   `@snapshot,recipe_name,snapshot`

   name - is the thumbdrive name which combines all the computer fields with '-'
   ip - [esxi-host reaper]:[port] (note, this port is the value observed when noting the Unique USB ID when setting up the USB on the ESXi Server. "esxi-host reaper" is the esxi host specified in reaper.rc.
   mac - forced "fake" mac address
   hytype - 'thumbdrive'
   pool - esxi-host reaper - (this value allows you to run Overmind tests so you can match VMs with USBs on the same esxi host)
   vlan - arbitrary
   reaper - 'nop' (specifies no clean state function for the thumbdrive resource entry)
   recipe_name - must match recipe_thumbs.csv
   snapshot - 'latest' (arbitrary)

   Example line in *computer_thumbs.csv* below:

```
td-fermions-2.0-thumbdrive-sandisk-cruzer-
16gb,fermions:2.0,00:50:56:us:b0:01,thumbdrive,fermions,thumb,nop
@snapshot,thumbdrive-sandisk-cruzer-td-16gb,latest
```

   NOTE: The IP attribute of the thumbdrive resource is used to reference the USB in testing and must be in the format [esxi-host reaper]:[port].

3. Import *recipes_thumbs.csv* by entering the command from tyworkflow:
   `bin/db_admin import_recipes recipes_thumbs.csv`

4. Import *computers_thumbs.csv* by entering the command from tyworkflow

```
bin/db_admin import_computers computers_thumbs.csv
```

5. Make sure to set the added thumbdrive(s) to use_testing in Overview.

9.4.3 Setup Tyrant Development environment

1. Add tyutils leafbag to tybase testing structure.
    a. Go to tybase/media directory.
    b. Create symbolic link to tyutils root directory:
    ```
    ln –s /proj/tyutils tyutils
    ```
    c. Go to tyworkflow directory.
    d. In tyworkflow root directory, call make_links.sh in tyutils. This creates a symbolic link to tyutils leafbag in tybase from the tyworkflow directory.
    ```
    ./media/tybase/media/tyutils/leafbag/make_links.sh
    ```

2. Setup *usb.rc* in tybase configuration.
    a. If usb.rc does not already exist in Tybase's rc directory, copy the existing *usb.rc* from *rc/default/usb.rc* to *rc/usb.rc*
    ```
    cp rc/default/usb.rc /rc
    ```
    b. Optionally edit the *usb.rc* file to set dirty file flags .

9.5 Tyrant USB API Functions

```
NAME
    tyutils.usb_utils

FILE
    /proj/tyutils/leafbag/tyutils/usb_utils.py

FUNCTIONS
    check_usb_blueprint(host, directory)
        Checks given walk directory against given usb's blueprint file
        Arguments:
        host - undermine host object
        directory - root directory

        Returns True if valid blueprint, False if invalid

    compare_usb_blueprint(host, directory, walk, remove_list=[])
        Checks given walk dictionary against given usb's blueprint file
        Arguments:
        host - undermine host object
        directory - root directory
        walk - compared walk with given directory

        Returns tuple (remove_list, size_diff_list) of differing files and sizes

        Raises exception upon error
        NOTE: walk is an output from calling walk_directory()

    connect_usb(host, usb, ip=None, disconnect_other=True)
        Connects given usb to given host on esxi server.
```

```
        Arguments:
        host - undermine host object to which usb will be connected
        usb - undermine usb object
        ip - (deprecated, do not use)
        disconnect_other - If True: if USB connected to another VM,
                will disconnect from previous host and connect it to this one.
                If False: will raise an exception.
                Default is True.

        Returns tuple (boolean,drive) True on success, False otherwise. For linux,
drive is mount path.

        Raises Exception upon error
        Note: Host and USB must be on the same ESXi server.

    create_usb_blueprint(host, directory, walk)
        Creates usb blueprint file for given usb, based on given walk dictionary
        Arguments:
        host - undermine host object
        directory - path to create blueprint
        walk - walk for blueprint

        Returns path to USB blueprint
        Raises exception upon error
        NOTE: walk is an output from calling walk_directory()

    disconnect_usb(host, usb, ip=None, vmname=None)
        Disconnects given usb from given host.
        Arguments:
        host - undermine host object to which usb will be connected
        usb - undermine usb object
        ip - (deprecated, do not use)
        vmname - (optional) virtual machine name to which usb will be connected
        Returns True on success, False otherwise

        Raises Exception upon error

    setup_usbs(host, usbs)
        Connects USB to a host and checks blueprint file for any dirty, unexpected
files.
        If setup fails, will raise exception
        Arguments:
        host - undermine host object to which usbs will be connected
        usbs - list of usbs to connect

    usb_remove_files(host, Files)
        Removes specified files from USB on given host
        Arguments:
        host - undermine host object
        Files - list of file paths to remove

    walk_directory(host, directory)
        Walks through the given directory on the given host and returns a dictionary
of files and their sizes
        Arguments:
        host - undermine host object
        directory - root directory to start walk

        Returns walk of nested dictionaries of directory sizes

DATA
    CONN_WAIT_MIN = 5
       Maximum wait time in minutes for USB to connect to Host
```

```
MOUNT_TIMEOUT_MIN = 1
   Maximum wait time in minutes for USB to show up on Host
WAIT_INTERVAL_SEC = 5
   Wait polling interval in seconds
```

9.6 Example Tyrant Test Scripts

Example tyrant test scripts using the USB APIs are also included in the Tyutils repository. These test scripts include *usb_test.py* and *usb_blueprint_test.py*.

In the example command line for test scripts in this section, hosts and a thumb drive have been added to Overmind with the following resource attributes (vlan, snapshot, lang, arch, and apps attributes are not included for brevity):

name	ip	mac	hwtype	pool	reaper	family	os	ossp
00-01-vm...	162.1.2.21	00:50:56:33:00:01	vm	p1	fermions	windows	2008 R2	0
00-02-vm...	162.1.2.22	00:50:56:33:00:02	vm	p1	fermions	windows	7	1
td-fermions...	fermions:1.2.0	00:50:56:us:b0:01	thumbdrive	fermions	nop	thumbdrive	sandisk	cruzer

9.6.1 usb_test.py

This test demonstrates basic functions with USBs. The test connects a USB to first host, writes a new file to USB, disconnects the USB, connects the USB to second host, checks if new file exists, deletes it, and finally disconnect the USB.

An example command line in Tybase to run this test with undermine:

```
bin/undermine tyutils.tests.usb_test 162.1.2.30 162.1.2.27 fermions:1.2.0
```

The code:

```
import time
import glob

import tybase.undermine.meta.leafi as leafi
import tybase.undermine.main_script as main_script
import tyutils.usb_utils as usb_utils

@leafi.MainLeaf()
class USBTest(main_script.Main_Script):
    def run(self):
        result = self.SUCCESS
        msg = 'Finished.'

        if len(self.usbs) < 1:
            return(self.SKIPPED,'Must specify one USB connected to an ESXi server.')

        if len(self.hosts) < 1:
            return(self.SKIPPED,'Must specify two hosts on the same ESXi server as USB.')

        usb = self.usbs[0]
        host1 = self.hosts[0]
        host2 = None
        if len(self.hosts) > 1:
            host2 = self.hosts[1]
```

```
        createFile = False
        added_filename = 'testfile.txt'

        try:
            usb_utils.disconnect_usb(host1,usb)
        except:
            pass
        try:
            if host2 is not None:
                usb_utils.disconnect_usb(host2,usb)
        except:
            pass

        try:


            ### EXAMPLE1: Simple Connect, write a file to it, list files, and disconnect
            # If 2nd host exists, connect to second host,
            # connect USB to host1
            # then glob the usb root directory, write another file to it, and disconnect.

            self.log.info('Connecting USB to',host1.ip)


            # NOTE: connect_usb defaults to disconnecting the usb if already connected to another
host.
            # when connects USB, if usb is connected to another unknown host, do NOT disconnect
it
            # Test will return "Attention" instead.
            rv, directory = usb_utils.connect_usb(host1,usb,disconnect_other=False)

            self.log.info('Globbing USB directory')
            glob_i = host1.mirrorfunc('import','glob')
            rv = glob_i.glob(directory+host1.sep()+'*')
            self.log.info('Glob result:',rv)

            # write a file on the thumbdrive
            bp_file = host1.mirrorfunc('open',directory+host1.sep()+added_filename,'w')
            bp_file.write('Hello, this is a test.\n')
            bp_file.write('This is only a test.\n')
            bp_file.close()

            createFile = True


            ### End EXAMPLE1

            try:
                self.log.info('Disconnecting USB from',host1.ip)
                usb_utils.disconnect_usb(host1,usb)
            except:
                pass


            if host2 is not None: #second host to test
                self.log.info('Sleeping 30 sec before connecting to next host.')
                time.sleep(30)
                self.log.info('Connecting USB to',host2.ip)
                rv, directory = usb_utils.connect_usb(host2,usb)


                self.log.info('Globbing USB directory')
                glob_i = host2.mirrorfunc('import','glob')
                rv = glob_i.glob(directory+host2.sep()+'*')
                self.log.info('Glob result:',rv)

                #delete file created from EXAMPLE1
                if createFile is True:
                    found = False
                    glob_file = directory+host2.sep()+added_filename
                    glob_file2 = glob_file.replace(host2.sep(),host2.sep()+host2.sep())
                    print "looking for",glob_file,'or',glob_file2
                    if glob_file in rv or glob_file2 in rv:
                        found = True
```

```
                    usb_utils.usb_remove_files(host2, [glob_file,glob_file2])
                if not found:
                    result = self.FAILURE
                    msg = 'File created not found.'

            self.log.info('Disconnecting USB from',host2.ip)
            usb_utils.disconnect_usb(host2,usb)

        except Exception, e:
            return (self.ATTENTION,'Error occured:'+str(e))

        return(result,msg)
```

9.6.2 usb_blueprint_test.py

This test connects a USB to first host, creates blueprint file on the USB, disconnects the USB, calls setup_usb() on the second host, and disconnects the USB.

An example command line in Tybase to run this test with undermine:

```
bin/undermine tyutils.tests.usb_blueprint_test 162.1.2.30 162.1.2.27 fermions:1.2.0
```

The code:

```
import time
import glob

import tybase.undermine.meta.leafi as leafi
import tybase.undermine.main_script as main_script
import tyutils.usb_utils as usb_utils

@leafi.MainLeaf()
class USBBlueprintTest(main_script.Main_Script):
    def run(self):
        result = self.SUCCESS
        msg = 'Finished.'

        SLEEP_MIN = 2
        if len(self.usbs) != 1:
            return(self.SKIPPED,'Must specify one USB connected to an ESXi server.')

        if len(self.hosts) < 2:
            return(self.SKIPPED,'Must specify two hosts on the same ESXi server as USB.')

        usb = self.usbs[0]
        host1 = self.hosts[0]
        host2 = self.hosts[1]


        try:
            usb_utils.disconnect_usb(host1,usb)
        except:
            pass
        try:
            usb_utils.disconnect_usb(host2,usb)
        except:
            pass


        try:


            ### EXAMPLE2: Create Blueprint file on thumbdrive
            # simply connect USB to host1,
            # then walk the usb root directory, and create (or replace) blueprint file on the USB
from that walk.


            self.log.info('Connecting USB to',host1.ip)
```

```
            rv, directory = usb_utils.connect_usb(host1,usb)

            self.log.info('Walking USB Directory')
            walk = usb_utils.walk_directory(host1, directory)
            self.log.info('Walk result:',walk)

            self.log.info('Creating blueprint file on USB')
            blueprint_path = usb_utils.create_usb_blueprint(host1, directory, walk)


            self.log.info('Disconnecting USB from',host1.ip)
            usb_utils.disconnect_usb(host1,usb)


            ### setup usbs
            # connect usb AND checks blueprint file. If blueprint file does not match contents,
            # will delete dirty files (or files not in blueprint),
            # REPEAT: if dirty files found, will delete them.
            try:
                time.sleep(5) #wait for full disconnect
                usb_utils.setup_usbs(host2, [usb])
            except Exception, e:
                return (self.FAILURE,'USBs Setup not successful:'+str(e))


            self.log.info('Sleeping',SLEEP_MIN,'min to manually check USB Connection')
            time.sleep(SLEEP_MIN*60)

        except Exception, e:
            return (self.ATTENTION,'Error occured:'+str(e))
        finally:
            try:
                self.log.info('Disconnecting USB from',host2.ip)
                usb_utils.disconnect_usb(host2,usb)
            except:
                pass

        return(result,msg)
```

9.6.3 Running Tests with Autoplan or Plan Files

To run a plan with usb tests, be sure to specify host slots that chose hosts and thumb drives connected to the same ESXi host. A technique to ensure this is to set thumb drive's Overmind resource's pool name to that of the computer's Overmind resource's reaper name.

In the example autoplan command line below, the host computer filter reaper value is the same as the host thumb drive pool value:

```
bin/autoplan solve tyutils.tests.usb_test -H reaper=fermions -H family=fermions -H
hwtype=thumbdrive,pool=fermions samples=2
```

This should generate the following plan file:

```
---
from tyworkflow.support.planlang import *

auto_tc = TESTCASE(
    script = 'tyutils.tests.usb_test',
    hostslots = [HOST(reaper='fermions'), HOST(reaper='fermions'),
HOST(hwtype='thumbdrive',pool='fermions')],
    samples = 2,
    planname = 'auto_20140304115730'
)

EXECUTE(
    testcase = auto_tc
```

)

This document includes install and setup instructions for the Eclipse IDE in reference to Dart Tyrant test development. The instructions in this document apply to the following system configurations:

- Fedora 20 64-bit operating system on a Gnome Desktop
- Eclipse 4.3.2 (Kepler)
- PyDev 3.4.1

10 Appendix F - Installing Eclipse IDE

There is always more than approach to install Eclipse for your Linux development environment.

10.1 With Yum

The instructions below outline two methods to install Eclipse 4.3.2 on Fedora 20 using Fedora's built in software package update and install.

1. Run the following command:
   ```
   yum install eclipse
   ```

   On a vanilla Fedora 20 operating system, the following lists the eclipse rpm install package and dependencies:

   ```
   Installed:
     eclipse-pde.x86_64 1:4.3.2-3.fc20

   Dependency Installed:
     ant.noarch 0:1.9.2-7.fc20
     ant-antlr.noarch 0:1.9.2-7.fc20
     ant-apache-bcel.noarch 0:1.9.2-7.fc20
     ant-apache-bsf.noarch 0:1.9.2-7.fc20
     ant-apache-log4j.noarch 0:1.9.2-7.fc20
     ant-apache-oro.noarch 0:1.9.2-7.fc20
     ant-apache-regexp.noarch 0:1.9.2-7.fc20
     ant-apache-resolver.noarch 0:1.9.2-7.fc20
     ant-apache-xalan2.noarch 0:1.9.2-7.fc20
     ant-commons-logging.noarch 0:1.9.2-7.fc20
     ant-commons-net.noarch 0:1.9.2-7.fc20
     ant-javamail.noarch 0:1.9.2-7.fc20
     ant-jdepend.noarch 0:1.9.2-7.fc20
     ant-jmf.noarch 0:1.9.2-7.fc20
     ant-jsch.noarch 0:1.9.2-7.fc20
     ant-junit.noarch 0:1.9.2-7.fc20
     ant-swing.noarch 0:1.9.2-7.fc20
     ant-testutil.noarch 0:1.9.2-7.fc20
     antlr-tool.noarch 0:2.7.7-29.fc20
     apache-commons-codec.noarch 0:1.8-5.fc20
     apache-commons-el.noarch 0:1.0-29.fc20
     apache-commons-logging.noarch 0:1.1.3-7.fc20
     apache-commons-net.noarch 0:3.3-2.fc20
   ```

```
atinject.noarch 0:1-13.20100611svn86.fc20
avalon-framework.noarch 0:4.3-9.fc20
avalon-logkit.noarch 0:2.1-13.fc20
batik.noarch 0:1.8-0.11.svn1230816.fc20
bcel.noarch 0:5.2-17.fc20
bea-stax-api.noarch 0:1.2.0-8.fc20
bsf.noarch 0:2.4.0-17.fc20
cglib.noarch 0:2.2-17.fc20
easymock.noarch 0:3.2-1.fc20
eclipse-ecf-core.noarch 0:3.8.0-1.fc20
eclipse-emf-core.noarch 1:2.9.2-1.fc20
eclipse-equinox-osgi.x86_64 1:4.3.2-3.fc20
eclipse-jdt.x86_64 1:4.3.2-3.fc20
eclipse-platform.x86_64 1:4.3.2-3.fc20
eclipse-swt.x86_64 1:4.3.2-3.fc20
felix-bundlerepository.noarch 0:1.6.6-15.fc20
felix-framework.noarch 0:4.2.1-4.fc20
felix-gogo-command.noarch 0:0.12.0-9.fc20
felix-gogo-runtime.noarch 0:0.10.0-10.fc20
felix-gogo-shell.noarch 0:0.10.0-9.fc20
felix-osgi-compendium.noarch 0:1.4.0-16.fc20
felix-osgi-core.noarch 0:1.4.0-14.fc20
felix-osgi-foundation.noarch 0:1.2.0-14.fc20
felix-osgi-obr.noarch 0:1.0.2-11.fc20
felix-shell.noarch 0:1.4.3-4.fc20
felix-utils.noarch 0:1.2.0-3.fc20
geronimo-annotation.noarch 0:1.0-14.fc20
geronimo-jms.noarch 0:1.1.1-17.fc20
glassfish-jsp.noarch 0:2.2.6-11.fc20
glassfish-jsp-api.noarch 0:2.2.1-8.fc20
hamcrest.noarch 0:1.3-5.fc20
httpcomponents-client.noarch 0:4.2.5-3.fc20
httpcomponents-core.noarch 0:4.2.4-5.fc20
icu4j.noarch 1:50.1.1-2.fc20
icu4j-eclipse.noarch 1:50.1.1-2.fc20
isorelax.noarch 1:0-0.14.release20050331.fc20
jai-imageio-core.noarch 0:1.2-0.13.20100217cvs.fc20
jakarta-oro.noarch 0:2.0.8-14.fc20
java-1.7.0-openjdk-devel.x86_64 1:1.7.0.60-2.4.7.0.fc20
java-1.7.0-openjdk-javadoc.noarch 1:1.7.0.60-2.4.7.0.fc20
javamail.noarch 0:1.5.0-6.fc20
jdepend.noarch 0:2.9.1-9.fc20
jetty-continuation.noarch 0:9.0.5-3.fc20
jetty-http.noarch 0:9.0.5-3.fc20
jetty-io.noarch 0:9.0.5-3.fc20
jetty-jmx.noarch 0:9.0.5-3.fc20
jetty-security.noarch 0:9.0.5-3.fc20
jetty-server.noarch 0:9.0.5-3.fc20
jetty-servlet.noarch 0:9.0.5-3.fc20
jetty-util.noarch 0:9.0.5-3.fc20
jsch.noarch 0:0.1.50-2.fc20
junit.noarch 0:4.11-7.fc20
```

```
  jvnet-parent.noarch 0:4-2.fc20
  jzlib.noarch 0:1.1.2-2.fc20
  kxml.noarch 0:2.3.0-3.fc20
  log4j.noarch 0:1.2.17-14.fc20
  lucene.noarch 0:3.6.2-3.fc20
  lucene-contrib.noarch 0:3.6.2-3.fc20
  mesa-libGLU.x86_64 0:9.0.0-4.fc20
  msv-xsdlib.noarch 1:2013.5.1-6.fc20
  objectweb-asm.noarch 0:3.3.1-8.fc20
  objenesis.noarch 0:1.2-16.fc20
  qdox.noarch 0:1.12.1-7.fc20
  regexp.noarch 0:1.5-13.fc20
  relaxngDatatype.noarch 0:1.0-11.5.fc20
  sac.noarch 0:1.3-17.fc20
  sat4j.noarch 0:2.3.5-2.fc20
  stax2-api.noarch 0:3.1.1-8.fc20
  tomcat-el-2.2-api.noarch 0:7.0.47-1.fc20
  tomcat-servlet-3.0-api.noarch 0:7.0.47-1.fc20
  webkitgtk.x86_64 0:2.2.7-1.fc20
  woodstox-core.noarch 0:4.2.0-2.fc20
  xalan-j2.noarch 0:2.7.1-22.fc20
  xerces-j2.noarch 0:2.11.0-16.fc20
  xml-commons-apis.noarch 0:1.4.01-14.fc20
  xml-commons-resolver.noarch 0:1.2-14.fc20
  xpp3.noarch 0:1.1.3.8-10.fc20

Dependency Updated:
  java-1.7.0-openjdk.x86_64 1:1.7.0.60-2.4.7.0.fc20
  java-1.7.0-openjdk-headless.x86_64 1:1.7.0.60-2.4.7.0.fc20
```

10.2 With the tar.gz file

These instructions come from the source:
http://www.if-not-true-then-false.com/2010/linux-install-eclipse-on-fedora-centos-red-hat-rhel/

1. Open a terminal and log in as Root user.
2. Extract the Eclipse package. (example to /opt directory)

```
## x86 - 32-bit ##
tar -xvzf eclipse-standard-kepler-R-linux-gtk.tar.gz -C /opt

## x86_64 - 64-bit ##
tar -xvzf eclipse-standard-kepler-R-linux-gtk-x86_64.tar.gz -C /opt
```

3. Add read permissions to all files.

```
chmod -R +r /opt/eclipse
```

4. Create Eclipse executable on /usr/bin path.

```
touch /usr/bin/eclipse
```

```
chmod 755 /usr/bin/eclipse
```

5. Open *eclipse* file with an editor.

   /usr/bin/eclipse

   And then paste following content to file:

```
#!/bin/sh
export ECLIPSE_HOME="/opt/eclipse"

$ECLIPSE_HOME/eclipse $*
```

6. Create Gnome desktop launcher. Create following file, with an editor:

   /usr/share/applications/eclipse.desktop

   And then add the following content to file and save:

```
[Desktop Entry]
Encoding=UTF-8
Name=Eclipse
Comment=Eclipse SDK 4.3.2
Exec=eclipse
Icon=/opt/eclipse/icon.xpm
Terminal=false
Type=Application
Categories=GNOME;Application;Development;
StartupNotify=true
```

10.3 Start Eclipse

From command line use eclipse command

```
eclipse
```

Or search for it in applications and click the Eclipse icon. (You can add it to the activity launcher bar by dragging the icon to it.)

10.4 Install Pydev

10.4.1 With Yum

1. Run the following command:
   `yum install eclipse-pydev`

   On a vanilla Fedora 20 operating system, the following lists the pydev rpm install package and dependencies:

```
Installed:
  eclipse-pydev.noarch 1:3.4.1-1.fc20
```

```
Dependency Installed:
  apache-commons-lang.noarch 0:2.6-13.fc20
  cal10n.noarch 0:0.7.7-3.fc20
  geronimo-jta.noarch 0:1.1.1-15.fc20
  javassist.noarch 0:3.16.1-6.fc20
  jpathwatch.x86_64 0:0.95-1.fc20
  jython.noarch 0:2.2.1-14.fc20
  libreadline-java.x86_64 0:0.8.0-33.fc20
  mysql-connector-java.noarch 1:5.1.28-1.fc20
  openpgm.x86_64 0:5.2.122-2.fc20
  pylint.noarch 0:1.1.0-1.fc20
  python-astroid.noarch 0:1.0.1-3.fc20
  python-backports.x86_64 0:1.0-3.fc20
  python-backports-ssl_match_hostname.noarch 0:3.4.0.2-1.fc20
  python-django.noarch 0:1.6.4-1.fc20
  python-django-bash-completion.noarch 0:1.6.4-1.fc20
  python-ipython-console.noarch 0:0.13.2-3.fc20
  python-logilab-common.noarch 0:0.61.0-1.fc20
  python-mglob.noarch 0:0.4-9.fc20
  python-pexpect.noarch 0:3.1-1.fc20
  python-setuptools.noarch 0:1.4.2-1.fc20
  python-simplegeneric.noarch 0:0.8-7.fc20
  python-zmq.x86_64 0:13.0.2-1.fc20
  slf4j.noarch 0:1.7.5-3.fc20
  ws-commons-util.noarch 0:1.0.1-27.fc20
  xmlrpc-client.noarch 1:3.1.3-7.fc20
  xmlrpc-common.noarch 1:3.1.3-7.fc20
  xmlrpc-server.noarch 1:3.1.3-7.fc20
  zeromq3.x86_64 0:3.2.4-1.fc20
```

10.4.2 With the zip file

1. Extract the contents of the zip file, *PyDev 3.4.1*, in the eclipse/dropins folder and restart Eclipse.

   ```
   unzip PyDev\ 3.4.1.zip
   ```

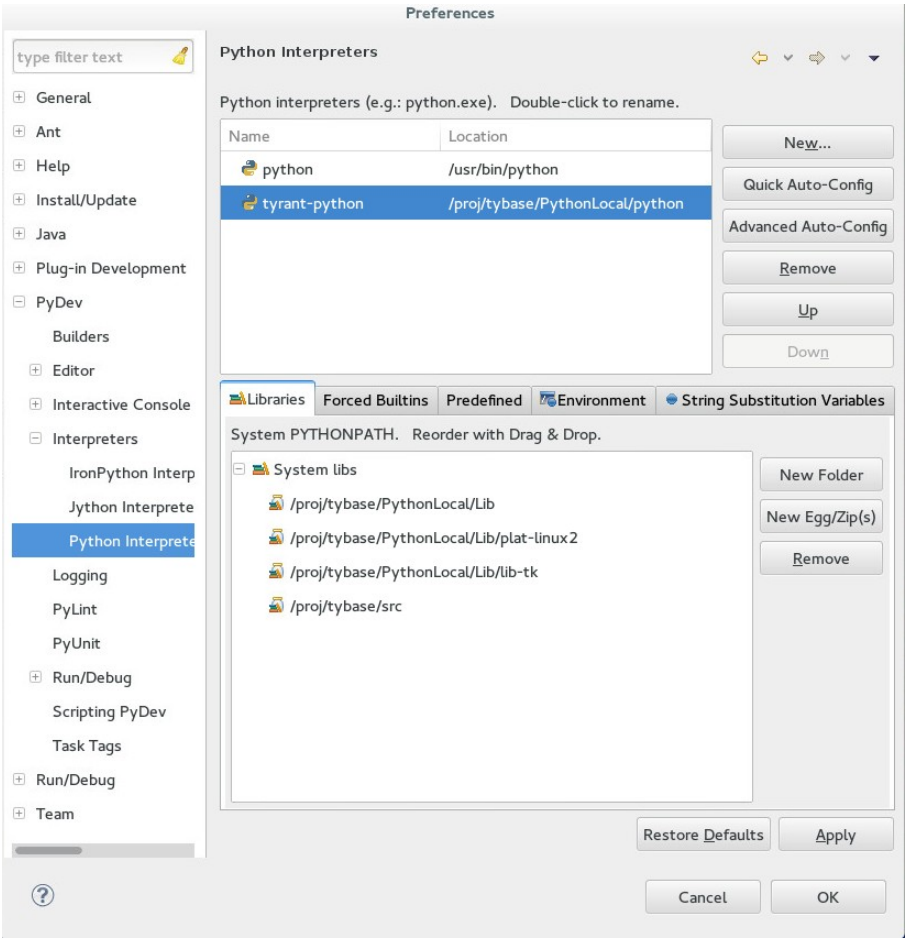
2. In Eclipse, setup a python interpreter for PyDev.
   a. Go to **Window > Preferences**
   b. Expand **PyDev > Interpreters > Python Interpreters**
   c. Click **Quick Auto-Config** to setup system python

10.5 Setup PyDev for Tyrant development

1. Create the Tyrant Python interpreter to be used for Tyrant development.
   a. Go to **Window > Preferences**
   b. Expand **PyDev > Interpreters > Python Interpreters**
   c. Click **New** for Tyrant's python.
   d. Give unique name "tyrant-python"
   e. Select executable at *[tybase]/PythonLocal/python* (where [tybase] is the root of Tyrant's *tybase* directory.

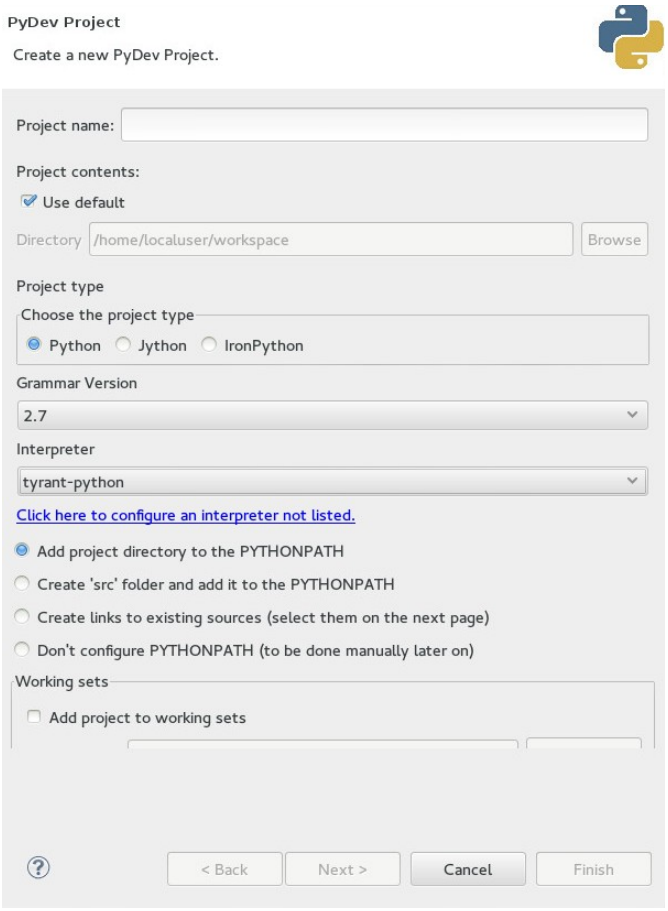
Add python path for code completion to tyrant libraries

f. Select the newly added Tyrant Python Interpreter, “tyrant-python”
g. In **Libraries** tab, click **New Folder**
h. Select path *[tybase]/src* and click **OK**.

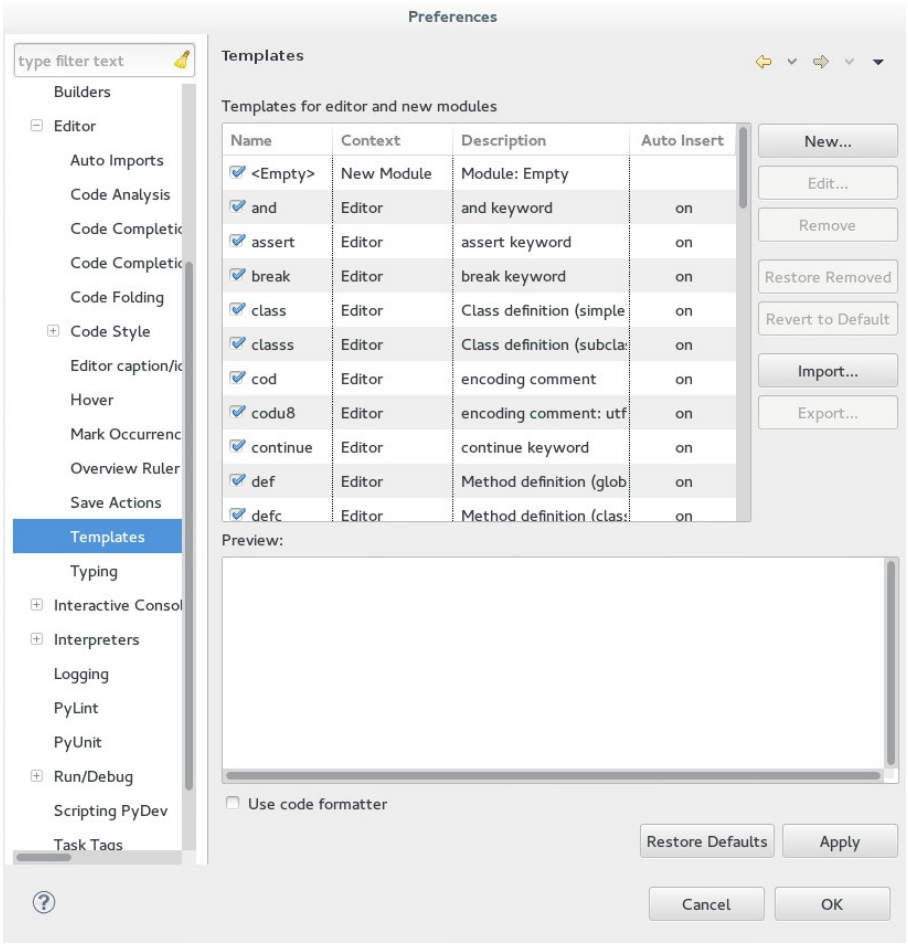


i. Click **Apply**. Allow the PYTHONPATH setup progress to complete.
j. Click **OK**.

Now when you create a new PyDev project for Tyrant, you must specify the “tyrant-python” interpreter in order to implement code completion and other context tip features for Tyrant libraries.



2. Import Tyrant module templates.
    a. Go to **Window > Preferences**
    b. Expand **PyDev > Editor > Templates**
    c. For each template in to import,
        i. Click **Import**.
        ii. Select an XML template in *[tybase]/docs/ide* and click **OK**.



d. Click **Apply**.
e. Click **OK**.

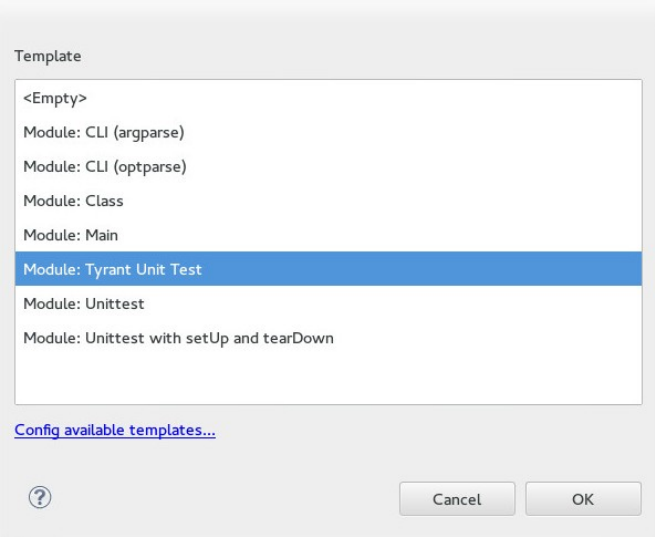
10.6 Code Completion and Context Tips

Once the PyDev environment is setup to include Tyrant libraries, you can take advantage of Tyrant templates, code completion and context tips when writing test scripts.

10.6.1 PyDev Module Templates

To create a new Tyrant test with an existing file template:

1. In a PyDev project, select the project and right click.
2. Select **New > PyDev Module**.
3. Enter the Name (and Folder and Package if necessary)
4. Click **Finish**.
5. Available PyDev Module Templates appear for you to choose, including the imported Tyrant templates.



6. Select *Module: Tyrant Unit Test* (or similar) template and click **OK**.

10.6.2 Viewing Tyrant Module Functions and Attributes

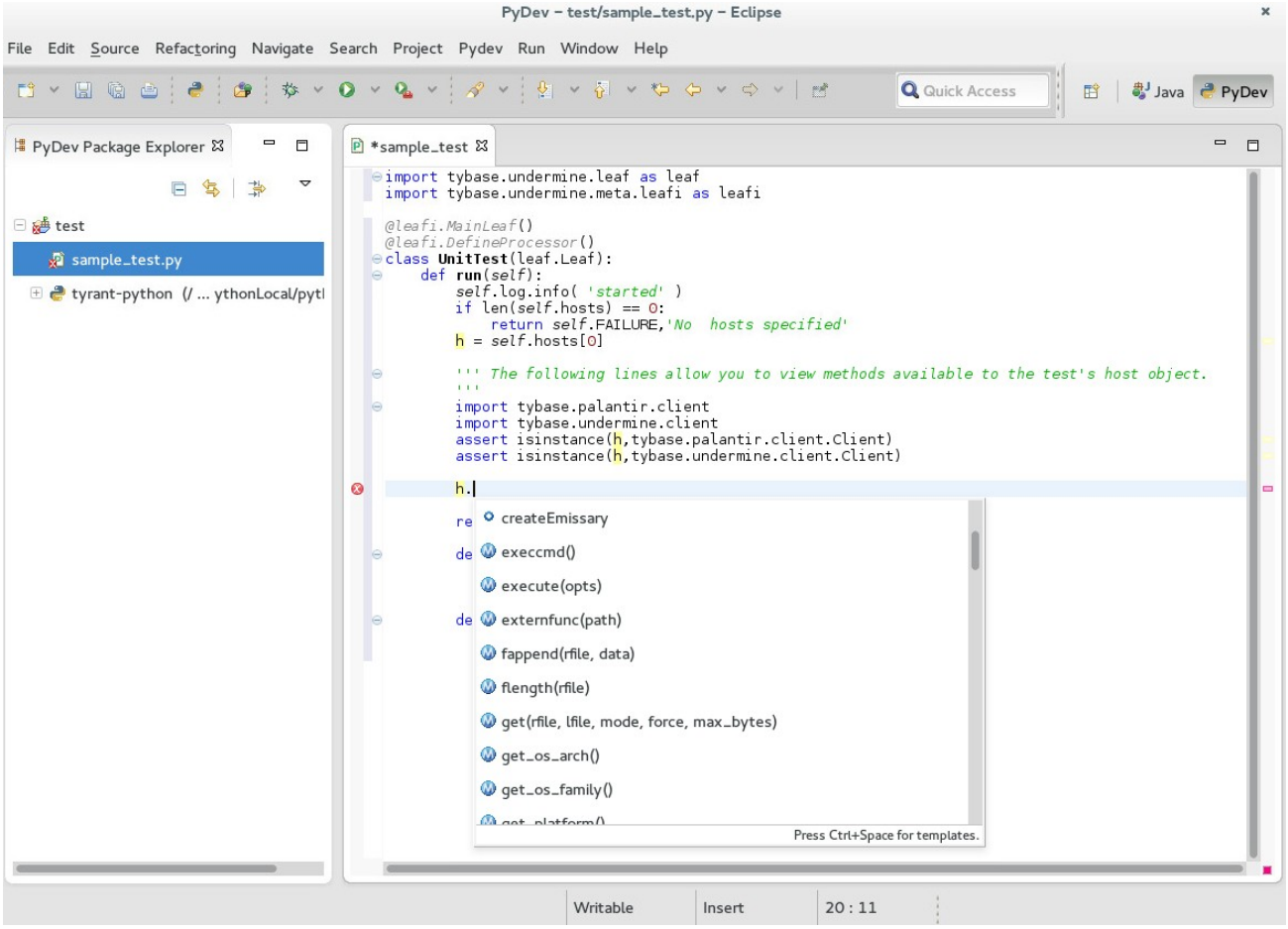
The "host" object is the most commonly manipulated object for Tyrant. The host object is a palantir Client instance. However, because these host objects are contained in python's type agnostic list data structure PyDev does not initially resolve the Client instance. To remedy this, you must import the base class modules and then call `assert isinstance` on the host object. The following code demonstrates this:

```
h = self.hosts[0]

import tybase.palantir.client
import tybase.undermine.client

assert isinstance(h,tybase.palantir.client.Client)
assert isinstance(h,tybase.undermine.client.Client)
```

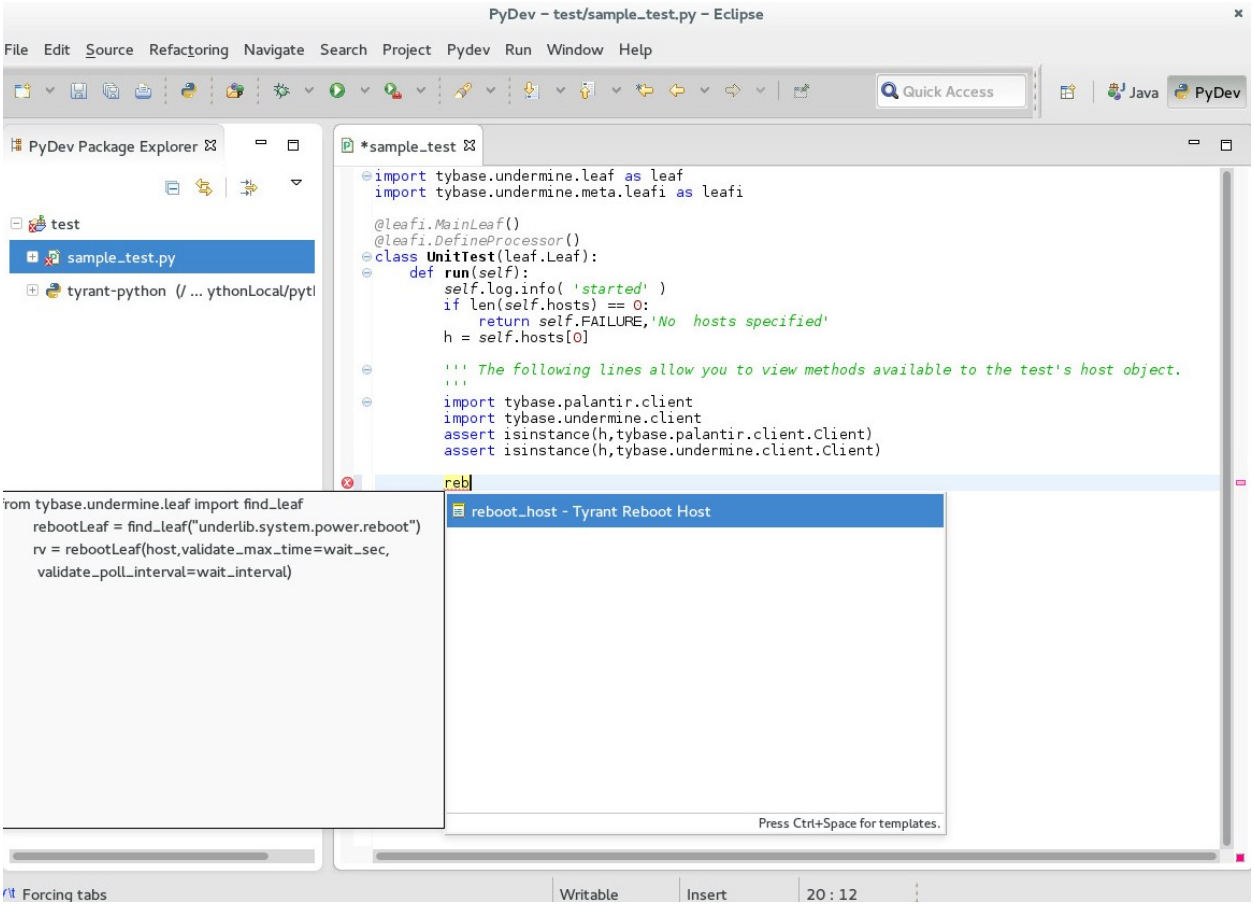
After adding the appropriate imports and assert statements to resolve the host instances, you can view methods available for a host object in the script.



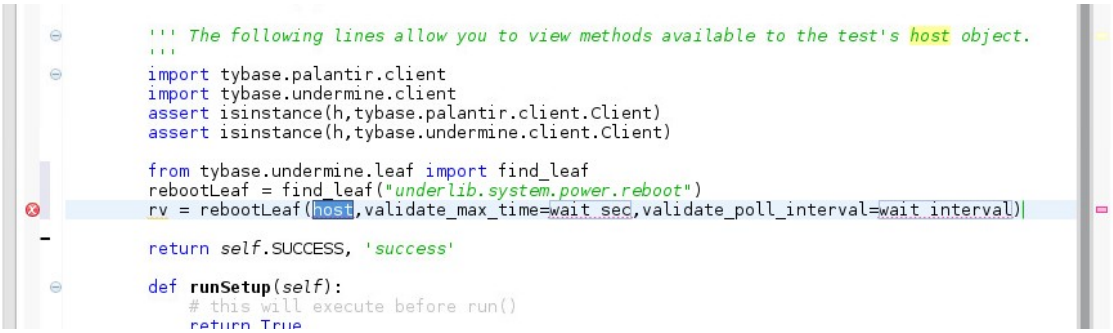
NOTE: If new modules are added or changes are made to the user added PYTHONPATH libs, you must refresh the Python Interpreter. To do this, open Python Interpreters, select the custom interpreter and click **Apply**.

10.6.3 PyDev Editor Templates

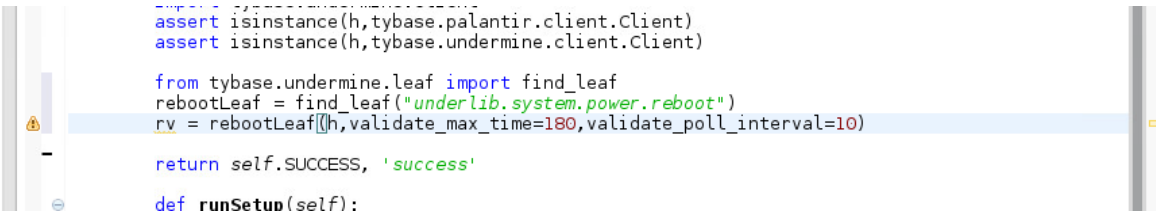
You can insert editor templates with the shortcut **CTRL+SPACE**. Then start typing the name of the editor template to filter before selecting the template to insert into code.



When the template you wish to insert is selected, press the **Enter** key. If there are any variables in the editor template, the first one will be highlighted.



Enter the appropriate variable for the code snippet and press **Tab** to go to the next variable. After filling in all variables, press **Enter** to complete code insertion.



11 Appendix G – Network Switching

In some test scenarios it may be necessary to cause an asset to switch from one network to another (and perhaps back again to the original one). Network switching functionality in Tyrant allows network switches to be performed programmatically during a test. An asset's network interface will be switched from the network it was originally connected to, to some other designated network. There, it will automatically be configured with a unique IP address valid for its new network. If the interface whose network is being changed is the interface over which palantir traffic (i.e. test control traffic) is traveling, then the palantir connection will be automatically reconfigured to work with the new network connection.

Network switching retrieves relevant information about original and new networks from the vlan table in the Tyrant database. Network switching is currently only supported for Windows VMs.

11.1 Range Setup Preconditions

Network switching requires some specialized range setup. Your range administrator should have done this for you, but here are a few quick items for reference in case you encounter problems:

- Each ESXi host in the range needs to be connected to any networks you wish to switch assets to.
- The name of a given network, if it is to be used for network switching, must be identical across ESXi servers.
- If you plan to switch the network of the interface which carries palantir traffic, then the location on which your tests are running (typically, the test server) must be able to access any networks you make the VMs switch to. In most cases, the interface carrying palantir traffic is the one that gets switched, so this condition likely applies to you.

11.2 Setup

Network switching is contained in the tyutils repository, so to use it, you must have tyutils linked in, as follows:

1. Clone the tyutils repository into the same directory in your testing environment (i.e. on your workstation) where tybase is located.
2. With the root of tybase as your current directory, run the following command:
   ```
   ln –s ../../tyutils media/tyutils
   ```
3. Change directory to the root of tyworkflow, then run the following command:
   ```
   ./media/tybase/media/tyutils/leafbag/make_links.sh
   ```

11.3 General Usage

Network switching is contained in the network_switch_utils module. Here we explain the general pattern of using network switching; details on how to call the functions come later.

First, you must import the module which provides network switching functionality, like so:

```
from tyutils import network_switch_utils
```

This module provides two functions: *network_switch* switches a given interface on a given asset to a given network and *end_switch* returns a single specified (or all) interface(s) on a given asset to their original network settings (with some caveats, explained later) and also unreserves any IP addresses which were previously reserved in the database for earlier calls to network_switch for the asset.

To perform network switching, call *network_switch* with the desired parameters (explained below). You may call *network_switch* multiple times, for the same interface or different interfaces, as needed, and the system will keep track of everything for you.

When you are done with network switching for an interface (or all interfaces), call *end_switch*. If, in your last call to *network_switch* for a particular interface on a particular asset, you did not dictate a specific IP address to use on the new network (and instead allowed the system to automatically pick for you), then you *must* call *end_switch*, because the call to *network_switch* caused an IP address to be reserved for you which will stay reserved. If you specified an IP address in your last call to *network_switch* for a particular interface on a particular asset, then calling *end_switch* is optional.

Like with other resource allocations which need to be cleaned up (such as allocated memory or multithreading locks), it is your responsibility when writing a test script to ensure that, if a call to *end_switch* is required, it is called even if your test script generates an error. You can use try/finally blocks to achieve this. Also note that since *end_switch* allows you to specify a specific interface to end network switching for, you must ensure that, if you do specify a specific interface at some point (instead of letting it work over all interfaces), you ultimately clean up network switching for all interfaces on an asset for which IP addresses where automatically selected and reserved. Typically, users allow the system to generate IP addresses for them and then clean everything up with a single call to *end_switch* to end network switching for all network interfaces on the asset, rather than calling *end_switch* individually for multiple interfaces.

11.4Warnings and Caveats

- As explained before, you must make sure that if IP addresses are automatically selected and reserved (due to not passing in an IP address to calls to *network_switch*), *end_switch* is called, even if an error occurs elsewhere in the script, so that IP addresses are unreserved.
- If a running testcase is purged (such as with the "bin/remote_commit purge" command) and that testcase had IP addresses reserved, those IP addresses will not be unreserved (because the testcase was abruptly killed due to the purge). We currently have no automatic way of handling this. Periodically, you will have to clean out old IP address reservations from the "ip" table in the tyrant database. Assuming that you have a single destination network used for network switching whose IP address range is nonoverlapping with the IP address range used by test assets' normal network connections, then as long as no tests are currently running which are using network switching, you can safely delete from the "ip" table all those IP addresses which are in that destination network range.

- When an asset's network interface is reconfigured as part of a call to *network_switch*, it is configured to use a static IP address on the new network. If that test asset is normally configured to use DHCP, *end_switch* is not capable of restoring the usage of DHCP. However, *end_switch* will restore the same IP address, subnet mask, and default gateway settings as the machine had prior to the switch; those settings will just be specified as static settings instead of received via DHCP. For automated Tyrant testing, this is sufficient, as the asset will eventually be reverted when it's used for a future test, and static configuration of the proper network settings is valid even on a network where DHCP is normally used.

11.5Function Details

The *network_switch* function is called as follows:

```
network_switch(host, network, ip, mask, gateway, ifnum,
      handle_palantir, timeout)
```

where the parameters are:

- host: Palantir client object connecting to the asset
- network (str): Name of the network you wish to connect to, which must match one of the networks on the ESXi host the asset resides on, and must match one of the records in the vlan table
- ip (str): IP address to use on the new network. This is optional; if not specified, a valid unique IP address is automatically picked from the IP address range in the database.
- mask (str): Subnet mask to use on the new network. This is option; if not specified, the subnet mask is fetched from the database vlan record.
- Ifnum (int): Index number (starting from 0) of the interface you wish to switch. This is optional and defaults to 0 (the first interface) if not specified.
- handle_palantir (bool): If True, then if the interface being switched is the one that carried palantir traffic, the palantir connection will automatically be updated to connect to the new IP address on the new network. Then, if this new connection fails to establish palantir connectivity with the asset, the network switch process will raise an exception. This is optional and defaults to True, which is almost certainly what you want.
- timeout (int): Maximum amount of time in seconds for the network switch to take place. This is optional and defaults to one minute.

Any parameters which are IP addresses are specified in IPv4 dotted-decimal format.

The *end_switch* function is called as follows:

```
end_switch(host, ifnum, handle_palantir, timeout)
```

where the parameters are:

- host: Palantir client object connecting to the asset

- ifnum (int): Index number (starting from 0) for the interface to end network switching on. If not specified, then network switching will be ended on all interfaces on the asset.
- handle_palantir (bool): Same meaning as with *network_switch*
- timeout (int): Same meaning as with *network_switch*