# SyncDET User Manual

April 17, 2012

## Contents



# 1 Introduction

SyncDET is a distributed experimentation and testing harness, intended to facilitate the execution of distributed software *scenarios*. The tool is launched from a *controller* system which deploys and synchronizes *actor* systems that play out the desired scenario.

In this user manual the system requirements and installation are covered, followed by an introduction to scenarios and the *test cases* embedded within them, concluding with instructions to write your own test cases.

# 2 Installation

## 2.1 System Requirements

Running SyncDET requires one or more UNIX, Linux, or cygwin systems. The controller has only been tested for OSX and requires the following:

1. Python 2.7+ installed
2. `git`, `tar`, and `grep`.

The requirements for each actor follow:

1. Python 2.6+ installed.
2. An `ssh` daemon server installed and running; future support can be made for `telnet` or `rlogin` if desired. The controller should be able to login without entering a password.
3. Commands `ps`, `grep`, `tar`, and `sed` installed. `ps` must support the option `-eo pid,cmd`, which lists all processes with a PID and a command line per row.

## 2.2 Acquiring Source Code

```
$ mkdir ~/repos; cd ~/repos
$ git clone ssh://g.arrowfs.org:44353/syncdet.git
```

## 2.3 Actor VMs

SyncDET-ready virtual machines have been created for Linux, OSX Leopard, Windows 7 and Windows XP. They are shared as bzipped tarballs in an AeroFS library somewhere (ask Mark or Weihan). They satisfy the system requirements listed above, except for passwordless ssh; **you must copy over your public key**. To extract the linux VM, for example:

```
$ tar -xvf syncdet-ubuntu1110_32.tar.bz2 -C <parent directory of VM>
```

## 2.4 Systems Configuration

On the controller system, first create a configuration file from the template:

```
$ sudo mkdir /etc/syncdet
$ cp syncdet/config.py.sample /etc/syncdet/config.py
```

Second, edit the configuration file to describe the controller and actors.

**Controller** You need to specify the hostname of the localhost from which you will launch SyncDET (the localhost will be the controller):

```
# the IP address or host name of the controller
controllerAddress = "192.168.1.15"
```

This is necessary because actors connect back to the controller for certain tasks.

**Actors** List the hostname or IP address of all systems to be used as actors:

```
# the dictionary definition of the actors
d_actors = [
            { 'address': '192.168.1.16' },
            { 'address': '192.168.1.17' },
            ]
```

### 2.5 Running the Demo

To run the entire scenario:

```
$ cd syncdet
$ ./syncdet.py examples/deploy -s examples/examples.scn
```

Or, to run a single test case:

```
$ ./syncdet.py examples/deploy -c examples.hello_world
```

## 3 Scenarios

From a controller (master) system, SyncDET deploys and executes distributed scenarios on multiple actor (slave) systems. This section explains the structure of a SyncDET scenario file, including the symbols and directives that constitute a scenario.

### 3.1 Sample Code

For an example, see (and follow) `examples/examples.scn`.

### 3.2 Concepts

A scenario essentially represents the combination of a number of execution *items*. An item can be a *test case* module, a group name, or even another scenario name. Test cases, the smallest building blocks of SyncDET scenarios, are small isolated Python functional tests that either pass or fail. Each item is executed concurrently on all actor systems, but *directives* are used to define the execution order of items with respect to one system. Items can be executed in some particular combination (e.g., sequentially or in random order), and frequently-used scenarios can be combined into larger scenarios.

Scenarios can be as simple or complex as you need; when writing your own SyncDET scenario, you may only require the use of one to three test cases.

## 3.3 Directives

`:serial`[,*count*] Execute items sequentially. Each item will be executed sequentially count times before proceeding to the next item. If count is zero, only the first item will be executed, and will be executed infinite times until the program is terminated. The default value for count is one. The two directives are interchangeable.

`:shuffle`[,*count*] Execute items sequentially, but in a random order. All items will be executed exactly once unless *count* is non-zero, in which case *count* items will be randomly selected and executed. If *count* is greater than the number of items, some items will be executed more than once. The default value for count is zero.

`:parallel`[,*count*] Execute all items in the block in parallel (with respect to one system). Each item will be copied count times and all copies will be executed in parallel with copies of other items. *count* must not be zero. Its default value is one.

`:scn[,nofail]` *name* Define a new scenario called *name*, which can be executed by calling *name*`()` from within a scenario file. With `nofail`, the entire scenario is stopped if any item fails, otherwise a failed item is skipped and the next item started.

`:include` *path* Link to scenario file *path* for defined scenarios.

`:group` *name* Define a group of test cases to which scenarios or other groups can refer by *name*`()`.

## 3.4 Use of Directives

In this section, we list some typical uses of the directives. To loop over a list of items for 10 times:

```
:serial,10
    :serial
        item1
        item2
        item3
```

To execute a random item in a list:

```
:shuffle,1
    item1
    item2
    item3
```

To randomly pick ten items from a list, and execute all of them in parallel:

```
    :parallel,10
        :shuffle,1
            item1
            item2
            item3
```

Note that in the above case, one item may have multiple instances executing in parallel. We can define a reusable scenario for the first example that will abort if any item fails, then execute it:

```
:scn,nofail firstEx
    :serial,10
        :serial
            item1
            item2
            item3

:serial
    firstEx()
```

# 4 Test Cases

In this section, we provide further details on writing test cases, the building blocks of scenarios. A test case is an ordinary Python script executed concurrently by some number of actor systems. Actors are assigned an ID giving them a total order. The following sections describe how to specify the point-of-entry of the test case, how to write test functions, and the various APIs to study.

## 4.1 Sample Code

For examples, see `hello_world.py` and `sync.py` in the demo code.

## 4.2 The `spec` data structure

Every test case python script must define a global dictionary, `spec`, which specifies the point-of-entry function to the test case across all actor systems, and its permissable duration. The following are the dictionary (key, value) pairs, and their meanings.

| key | value |
|---|---|
| `'entries'` | a list of length $n$, specifying the entry-point function names for the first $n$ systems |
| `'default'` | the name of the entry-point function name for all other systems |
| `'timeout'` | the maximum execution time of the test case, after which the test fails (integer, in seconds) |

At least one of `entries` or `default` are required, and `timeout` is entirely optional, with the default specified in `config.py`

## 4.3 How test failures are detected

A test case is considered a failure and will be reported, in any of the following situations:

- An exception is raised and not caught within the entry-point function.
- The test case process terminates before the entry-point function returns.
- The entry-point function doesn't complete in time. (See CASE_TIMEOUT in config.py for details.)

Therefore, the simplest way to fail a test case is to raise an arbitrary exception.

## 4.4 Test Case API: `import case`

This section briefly describes the general SyncDET API for writing test cases.

For functions to get information about the currently executing test case or scenario see `[SyncDETRoot]/syncdet/case/syncdet_case_lib.py`:

- get the local system ID,
- get the total number of systems executing the given test case,
- get the local SyncDET root (on the actor)
- get the test case module name, and more.

Optionally, test-case-writers could have access to the actors.Actor class, providing methods to scp files/folders to other actor systems. This is not officially supported yet, but given any requests, it can be arranged.

### 4.4.1 Synchronization API

Three functions of interest are defined in `[SyncDETRoot]/syncdet/case/syncdet_case_sync.py`:

- `case.sync(...)`, a barrier across all systems
- `case.syncPrev(...)`, a barrier shared with this system and the previous
- `case.syncNext(...)`, a barrier shared with this system and the next

This API is also accessed through the `case` module.

## 4.5 AeroFS-specific Test API (`aft` directory)

Write your AeroFS-specific test scenarios and cases in the `aft` directory.

**Python** An API of python functions in `[SyncDETRoot]/aft/lib/lib.py` will:

- get AeroFS directories: Application, RunTime, File-sharing mount
- launch AeroFS (assuming it is installed)
- kill/terminate AeroFS
- create a directory tree of files

Simply `import lib` from your test cases

**Scenario** Also consider using the `common.scn` scenario file for a Scenario interface to

- `start` AeroFS
- `stop` AeroFS

For both APIs, more functions will be written as required.