Ducktape Documentation

Release 0.7.23

Confluent Inc.

Mar 31, 2023

Contents

Ducktape contains tools for running system integration and performance tests. It provides the following features:

- Write tests for distributed systems in a simple unit test-like style
- Isolation by default so system tests are as reliable as possible.
- Utilities for pulling up and tearing down services easily in clusters in different environments (e.g. local, custom cluster, Vagrant, K8s, Mesos, Docker, cloud providers, etc.)
- Trigger special events (e.g. bouncing a service)
- Collect results (e.g. logs, console output)
- Report results (e.g. expected conditions met, performance results, etc.)

CHAPTER 1

Install

1. Install cryptography (used by *paramiko* which Ducktape depends on), this may have non-python external requirements

- OSX (if needed):

```
brew install openssl
```

- Ubuntu:

```
sudo apt-get install build-essential libssl-dev libffi-dev python-dev
```

- Fedora and RHEL-derivatives:

```
sudo yum install gcc libffi-devel python-devel openssl-devel
```

2. As a general rule, it's recommended to use an isolation tool such as `virtualenv`
3. Install Ducktape:

```
pip install ducktape
```

Note: On OSX you may need to:

```
C_INCLUDE_PATH=/usr/local/opt/openssl/include LIBRARY_PATH=/usr/local/opt/openssl/lib␣
↪pip install ducktape
```

If you are not using a virtualenv and get the error message *failed with error code 1*, you may need to install ducktape to your user directory instead with

```
pip install --user ducktape
```

CHAPTER 2

Test Clusters

Ducktape runs on a test cluster with several nodes. Ducktape will take ownership of the nodes and handle starting, stopping, and running services on them.

Many test environments are possible. The nodes may be local nodes, running inside Docker. Or they could be virtual machines running on a public cloud.

2.1 Cluster Specifications

A cluster specification– also called a ClusterSpec– describes a particular cluster configuration. Currently the cluster specification can express the number of nodes of each operating system that are required.

Cluster specifications give us a vocabulary to express what a particular service or test needs to run. For example, a service might require a cluster with three Linux nodes and one Windows node. We could express that with a ClusterSpec containing three Linux NodeSpec objects and one Windows NodeSpec object.

CHAPTER 3

Run Tests

3.1 Running Tests

ducktape discovers and runs tests in the path provided, here are some ways to run tests:

```
ducktape <relative_path_to_testdirectory>               # e.g. ducktape dir/tests
ducktape <relative_path_to_file>                        # e.g. ducktape dir/tests/my_
↪test.py
ducktape <path_to_test>[::SomeTestClass]                # e.g. ducktape dir/tests/my_
↪test.py::TestA
ducktape <path_to_test>[::SomeTestClass[.test_method]]  # e.g. ducktape dir/tests/my_
↪test.py::TestA.test_a
```

3.2 Options

To see a complete listing of options run:

```
ducktape --help
```

Discover and run your tests

```
usage: ducktape [-h] [--collect-only] [--collect-num-nodes] [--debug]
                [--config-file CONFIG_FILE] [--compress] [--cluster CLUSTER]
                [--default-num-nodes DEFAULT_NUM_NODES]
                [--cluster-file CLUSTER_FILE] [--results-root RESULTS_ROOT]
                [--exit-first] [--no-teardown] [--version]
                [--parameters PARAMETERS] [--globals GLOBALS]
                [--max-parallel MAX_PARALLEL] [--repeat REPEAT]
                [--subsets SUBSETS] [--subset SUBSET]
                [--historical-report HISTORICAL_REPORT] [--sample SAMPLE]
                [--fail-bad-cluster-utilization]
                [--test-runner-timeout TEST_RUNNER_TIMEOUT]
                [--ssh-checker-function SSH_CHECKER_FUNCTION [SSH_CHECKER_FUNCTION ...
↪]]
                [--deflake DEFLAKE]
                [test_path [test_path ...]]
```

Required Arguments

test_path=['/home/docs/checkouts/readthedocs.org/user_builds/ducktape/checkouts/0.7.x/docs'] one or more space-delimited strings indicating where to search for tests.

Optional Arguments

--collect-only=False display collected tests, but do not run.

--collect-num-nodes=False display total number of nodes requested by all tests, but do not run anything.

--debug=False pipe more verbose test output to stdout.

--config-file="~/.ducktape/config" path to project-specific configuration file.

--compress=False compress remote logs before collection.

--cluster="ducktape.cluster.vagrant.VagrantCluster" cluster class to use to allocate nodes for tests.

--default-num-nodes Global hint for cluster usage. A test without the @cluster annotation will default to this value for expected cluster usage.

--cluster-file path to a json file which provides information needed to initialize a json cluster. The file is used to read/write cached cluster info if cluster is ducktape.cluster.vagrant.VagrantCluster.

--results-root="./results" path to custom root results directory. Running ducktape with this root specified will result in new test results being stored in a subdirectory of this root directory.

--exit-first=False exit after first failure

--no-teardown=False don't kill running processes or remove log files when a test has finished running. This is primarily useful for test developers who want to interact with running services after a test has run.

--version=False display version

--parameters inject these arguments into the specified test(s). Specify parameters as a JSON string.

--globals user-defined globals go here. This can be a file containing a JSON object, or a string representing a JSON object.

--max-parallel=1 Upper bound on number of tests run simultaneously.

--repeat=1 Use this flag to repeat all discovered tests the given number of times.

--subsets=1 Number of subsets of tests to statically break the tests into to allow for parallel execution without coordination between test runner processes.

--subset=0 Which subset of the tests to run, based on the breakdown using the parameter for –subsets

--historical-report URL of a JSON report file containing stats from a previous test run. If specified, this will be used when creating subsets of tests to divide evenly by total run time instead of by number of tests.

--sample	The size of a random test sample to run
--fail-bad-cluster-utilization=False	Fail a test if the cluster node utilization does not match the cluster node usage.
--test-runner-timeout=1800000	Amount of time in milliseconds between test communicating between the test runner before a timeout error occurs. Default is 30 minutes
--ssh-checker-function	Python module path(s) to a function that takes an exception and a remote account that will be called when an ssh error occurs, this can give some validation or better logging when an ssh error occurs. Specify any number of module paths after this flag to be called.
--deflake=1	the number of times a failed test should be ran in total (including its initial run) to determine flakyness. When not present, deflake will not be used, and a test will be marked as either passed or failed. When enabled tests will be marked as flaky if it passes on any of the reruns

3.3 Configuration File

You can configure options in three locations: on the command line (highest priority), in a user configuration file in `~/.ducktape/config`, and in a project-specific configuration `<project_dir>/.ducktape/config` (lowest priority). Configuration files use the same syntax as command line arguments and may split arguments across multiple lines:

```
--debug
--exit-first
--cluster=ducktape.cluster.json.JsonCluster
```

3.4 Output

Test results go in `results/<session_id>.<session_id>` which looks like `<date>--<test_number>`. For example: `results/2015-03-28--002`

ducktape does its best to group test results and log files in a sensible way. The output directory is structured like so:

```
<session_id>
    session_log.info
    session_log.debug
    report.txt    # Summary report of all tests run in this session
    report.html   # Open this to see summary report in a browser
    report.css

    <test_class_name>
        <test_method_name>
            test_log.info
            test_log.debug
            report.txt    # Report on this single test
            [data.json]   # Present if the test returns data

            <service_1>
                <node_1>
                    some_logs
                <node_2>
```

```
                        some_logs
    ...
```

To see an example of the output structure, go here and click on one of the details links.

CHAPTER 4

Create New Tests

4.1 Writing ducktape Tests

Subclass `Test` and implement as many `test` methods as you want. The name of each test method must start or end with `test`, e.g. `test_functionality` or `example_test`. Typically, a test will start a few services, collect and/or validate some data, and then finish.

If the test method finishes with no exceptions, the test is recorded as successful, otherwise it is recorded as a failure.

Here is an example of a test that just starts a Zookeeper cluster with 2 nodes, and a Kafka cluster with 3 nodes:

```
class StartServicesTest(Test):
    """Make sure we can start Kafka and Zookeeper services."""
    def __init__(self, test_context):
        super(StartServicesTest, self).__init__(test_context=test_context)
        self.zk = ZookeeperService(test_context, num_nodes=2)
        self.kafka = KafkaService(test_context, num_nodes=3, self.zk)

    def test_services_start(self):
        self.zk.start()
        self.kafka.start()
```

4.2 Test Parameters

Use test decorators to parametrize tests, examples are provided below

`ducktape.mark.`**`parametrize`**`(`***kwargs*`)`

Function decorator used to parametrize its arguments. Decorating a function or method with `@parametrize` marks it with the Parametrize mark.

Example:

```
@parametrize(x=1, y=2 z=-1)
@parametrize(x=3, y=4, z=5)
def g(x, y, z):
    print "x = %s, y = %s, z = %s" % (x, y, z)

for ctx in MarkedFunctionExpander(..., function=g, ...).expand():
    ctx.function()

# output:
# x = 1, y = 2, z = -1
# x = 3, y = 4, z = 5
```

`ducktape.mark.`**`matrix`**`(`***kwargs*`)`

Function decorator used to parametrize with a matrix of values. Decorating a function or method with `@matrix` marks it with the Matrix mark. When expanded using the `MarkedFunctionExpander`, it yields a list of TestContext objects, one for every possible combination of arguments.

Example:

```
@matrix(x=[1, 2], y=[-1, -2])
def g(x, y):
    print "x = %s, y = %s" % (x, y)

for ctx in MarkedFunctionExpander(..., function=g, ...).expand():
    ctx.function()

# output:
# x = 1, y = -1
# x = 1, y = -2
# x = 2, y = -1
# x = 2, y = -2
```

`ducktape.mark.resource.`**`cluster`**`(`***kwargs*`)`

Test method decorator used to provide hints about how the test will use the given cluster.

Keywords used by ducktape

- `num_nodes` provide hint about how many nodes the test will consume
- `cluster_spec` provide hint about how many nodes of each type the test will consume

Example:

```
# basic usage with num_nodes
@cluster(num_nodes=10)
def the_test(...):
    ...

# basic usage with cluster_spec
@cluster(cluster_spec=ClusterSpec.simple_linux(10))
def the_test(...):
    ...

# parametrized test:
# both test cases will be marked with cluster_size of 200
@cluster(num_nodes=200)
@parametrize(x=1)
@parametrize(x=2)
def the_test(x):
    ...

# test case {'x': 1} has cluster size 100, test case {'x': 2} has cluster size 200
@cluster(num_nodes=100)
@parametrize(x=1)
@cluster(num_nodes=200)
@parametrize(x=2)
def the_test(x):
    ...
```

`ducktape.mark.ignore`(**args*, ***kwargs*)
Test method decorator which signals to the test runner to ignore a given test.

Example:

```
When no parameters are provided to the @ignore decorator, ignore all␣
↪parametrizations of the test function

@ignore  # Ignore all parametrizations
@parametrize(x=1, y=0)
@parametrize(x=2, y=3)
def the_test(...):
    ...
```

Example:

```
If parameters are supplied to the @ignore decorator, only ignore the␣
↪parametrization with matching parameter(s)

@ignore(x=2, y=3)
@parametrize(x=1, y=0)  # This test will run as usual
@parametrize(x=2, y=3)  # This test will be ignored
def the_test(...):
    ...
```

4.3 Logging

The `Test` base class sets up logger you can use which is tagged by class name, so adding some logging for debugging or to track the progress of tests is easy:

```
self.logger.debug("End-to-end latency %d: %s", idx, line.strip())
```

These types of tests can be difficult to debug, so err toward more rather than less logging.

Note: Logs are collected a multiple log levels, and only higher log levels are displayed to the console while the test runs. Make sure you log at the appropriate level.

4.4 New test example

Lets expand on the StartServicesTest example. The test starts a Zookeeper cluster with 2 nodes, and a Kafka cluster with 3 nodes, and then bounces a kafka broker node which is either a special controller node or a non-controller node, depending on the *bounce_controller_broker* test parameter.

```
class StartServicesTest(Test):
    def __init__(self, test_context):
        super(StartServicesTest, self).__init__(test_context=test_context)
        self.zk = ZookeeperService(test_context, num_nodes=2)
        self.kafka = KafkaService(self.test_context, num_nodes=3, zk=self.zk)

    def setUp(self):
        self.zk.start()
        self.kafka.start()

    @matrix(bounce_controller_broker=[True, False])
    def test_broker_bounce(self, bounce_controller_broker=False):
        controller_node = self.kafka.controller()
        self.logger.debug("Found controller broker %s", controller_node.account)
        if bounce_controller_broker:
            bounce_node = controller_node
        else:
            bounce_node = self.kafka.nodes[(self.kafka.idx(controller_node) + 1) %␣
↪self.kafka.num_nodes]

        self.logger.debug("Will hard kill broker %s", bounce_node.account)
        self.kafka.signal_node(bounce_node, sig=signal.SIGKILL)

        wait_until(lambda: not self.kafka.is_registered(bounce_node),
                   timeout_sec=self.kafka.zk_session_timeout + 5,
                   err_msg="Failed to see timely deregistration of hard-killed broker
↪%s"
                           % bounce_node.account)

        self.kafka.start_node(bounce_node)
```

This will run two tests, one with 'bounce_controller_broker': False and another with 'bounce_controller_broker': True arguments. We moved start of Zookeeper and Kafka services to `setUp()`, which is called before every test run.

The test finds which of Kafka broker nodes is a special controller node via provided `controller` method in KafkaService. The `controller` method in KafkaService will raise an exception if the controller node is not found. Make sure to check the behavior of methods provided by a service or other helper classes and fail the test as soon as an issue is found. That way, it will be much easier to find the cause of the test failure.

The test then finds the node to bounce based on *bounce_controller_broker* test parameter and then forcefully terminates the service process on that node via `signal_node` method of KafkaService. This method just sends a signal to forcefully kill the process, and does not do any further check. Thus, our test needs to check that the hard killed kafka broker is not part of the Kafka cluster anymore, before restarting the killed broker process. We do this by waiting on `is_registered` method provided by KafkaService to return False with a timeout, since de-registering the broker may take some time. Notice the use of `wait_until` method instead of a check after `time.sleep`. This allows the test to continue as soon as de-registration happens.

We don't check if the restarted broker is registered, because this is already done in KafkaService `start_node` implementation, which will raise an exception if the service is not started successfully on a given node.

CHAPTER 5

Create New Services

5.1 Writing ducktape services

`Service` refers generally to multiple processes, possibly long-running, which you want to run on the test cluster.

These can be services you would actually deploy (e.g., Kafka brokers, ZK servers, REST proxy) or processes used during testing (e.g. producer/consumer performance processes). Services that are distributed systems can support a variable number of nodes which allow them to handle a variety of tests.

Each service is implemented as a class and should at least implement the following:

- `start_node()` - start the service (possibly waiting to ensure it started successfully)
- `stop_node()` - kill processes on the given node
- `clean_node()` - remove persistent state leftover from testing, e.g. log files

These may block to ensure services start or stop properly, but must *not* block for the full lifetime of the service. If you need to run a blocking process (e.g. run a process via SSH and iterate over its output), this should be done in a background thread. For services that exit after completing a fixed operation (e.g. produce N messages to topic foo), you should also implement `wait`, which will usually just wait for background worker threads to exit. The `Service` base class provides a helper method `run` which wraps `start`, `wait`, and `stop` for tests that need to start a service and wait for it to finish. You can also provide additional helper methods for common test functionality. Normal services might provide a `bounce` method.

Most of the code you'll write for a service will just be series of SSH commands and tests of output. You should request the number of nodes you'll need using the `num_nodes` or `cluster_spec` parameter to the Service base class's constructor. Then, in your Service's methods you'll have access to `self.nodes` to access the nodes allocated to your service. Each node has an associated `RemoteAccount` instance which lets you easily perform remote operations such as running commands via SSH or creating files. By default, these operations try to hide output (but provide it to you if you need to extract some subset of it) and *checks status codes for errors* so any operations that fail cause an obvious failure of the entire test.

5.2 New Service Example

Let's walk through an example of writing a simple Zookeeper service.

```
class ZookeeperService(Service):
    PERSISTENT_ROOT = "/mnt"
    LOG_FILE = os.path.join(PERSISTENT_ROOT, "zk.log")
    DATA_DIR = os.path.join(PERSISTENT_ROOT, "zookeeper")
    CONFIG_FILE = os.path.join(PERSISTENT_ROOT, "zookeeper.properties")

    logs = {
        "zk_log": {
            "path": LOG_FILE,
            "collect_default": True},
        "zk_data": {
            "path": DATA_DIR,
            "collect_default": False}
    }

    def __init__(self, context, num_nodes):
        super(ZookeeperService, self).__init__(context, num_nodes)
```

`logs` is a member of `Service` that provides a mechanism for locating and collecting log files produced by the service on its nodes. `logs` is a dict with entries that look like `log_name: {"path": log_path, "collect_default": boolean}`. In our example, log files will be collected on both successful and failed test runs, while files from the data directory will be collected only on failed test runs. Zookeeper service requests the number of nodes passed to its constructor by passing `num_nodes` parameters to the Service base class's constructor.

```
def start_node(self, node):
    idx = self.idx(node)
    self.logger.info("Starting ZK node %d on %s", idx, node.account.hostname)

    node.account.ssh("mkdir -p %s" % self.DATA_DIR)
    node.account.ssh("echo %d > %s/myid" % (idx, self.DATA_DIR))

    prop_file = """\n dataDir=%s\n clientPort=2181""" % self.DATA_DIR
    for idx, node in enumerate(self.nodes):
        prop_file += "\n server.%d=%s:2888:3888" % (idx, node.account.hostname)
    self.logger.info("zookeeper.properties: %s" % prop_file)
    node.account.create_file(self.CONFIG_FILE, prop_file)

    start_cmd = "/opt/kafka/bin/zookeeper-server-start.sh %s 1>> %s 2>> %s &" % \
            (self.CONFIG_FILE, self.LOG_FILE, self.LOG_FILE)

    with node.account.monitor_log(self.LOG_FILE) as monitor:
        node.account.ssh(start_cmd)
        monitor.wait_until(
            "binding to port",
            timeout_sec=100,
            backoff_sec=7,
            err_msg="Zookeeper service didn't finish startup"
        )
    self.logger.debug("Zookeeper service is successfully started.")
```

The `start_node` method first creates directories and the config file on the given node, and then invokes the start script to start a Zookeeper service. In this simple example, the config file is created from manually constructed `prop_file` string, because it has only a couple of easy to construct lines. More complex config files can be created with templates, as described in *Using Templates*.

A service may take time to start and get to a usable state. Using sleeps to wait for a service to start often leads to a flaky test. The sleep time may be too short, or the service may fail to start altogether. It is useful to verify that the service starts properly before returning from the `start_node`, and fail the test if the service fails to start. Otherwise, the test will likely fail later, and it would be harder to find the root cause of the failure. One way to check that the service starts successfully is to check whether a service's process is alive and one additional check that the service is usable such as querying the service or checking some metrics if they are available. Our example checks whether a Zookeeper service is started successfully by searching for a particular output in a log file.

The `RemoteAccount` instance associated with each node provides you with `LogMonitor` that let you check or wait for a pattern to appear in the log. Our example waits for 100 seconds for "binding to port" string to appear in the `self.LOG_FILE` log file, and raises an exception if it does not.

```
def pids(self, node):
    try:
        cmd = "ps ax | grep -i zookeeper | grep java | grep -v grep | awk '{print $1}'
↪"
        pid_arr = [pid for pid in node.account.ssh_capture(cmd, allow_fail=True,␣
↪callback=int)]
        return pid_arr
    except (RemoteCommandError, ValueError) as e:
        return []

def alive(self, node):
    return len(self.pids(node)) > 0

def stop_node(self, node):
    idx = self.idx(node)
    self.logger.info("Stopping %s node %d on %s" % (type(self).__name__, idx, node.
↪account.hostname))
    node.account.kill_process("zookeeper", allow_fail=False)

def clean_node(self, node):
    self.logger.info("Cleaning Zookeeper node %d on %s", self.idx(node), node.account.
↪hostname)
    if self.alive(node):
        self.logger.warn("%s %s was still alive at cleanup time. Killing forcefully...
↪" %
                         (self.__class__.__name__, node.account))
    node.account.kill_process("zookeeper", clean_shutdown=False, allow_fail=True)
    node.account.ssh("rm -rf /mnt/zookeeper /mnt/zookeeper.properties /mnt/zk.log",
                     allow_fail=False)
```

The `stop_node` method uses `kill_process()` to terminate the service process on the given node. If the remote command to terminate the process fails, `kill_process()` will raise an `RemoteCommandError` exception.

The `clean_node` method forcefully kills the process if it is still alive, and then removes persistent state leftover from testing. Make sure to properly cleanup the state to avoid test order dependency and flaky tests. You can assume complete control of the machine, so it is safe to delete an entire temporary working space and kill all java processes, etc.

5.3 Using Templates

Both `Service` and `Test` subclass `TemplateRenderer` that lets you render templates directly from strings or from files loaded from *templates/* directory relative to the class. A template contains variables and/or expressions, which are replaced with values when a template is rendered. `TemplateRenderer` renders templates using Jinja2 template engine. A good use-case for templates is a properties file that needs to be passed to a service process. In *New Service Example*, the properties file is created by building a string and using it as contents as follows:

```
prop_file = """\n dataDir=%s\n clientPort=2181""" % self.DATA_DIR
for idx, node in enumerate(self.nodes):
    prop_file += "\n server.%d=%s:2888:3888" % (idx, node.account.hostname)
node.account.create_file(self.CONFIG_FILE, prop_file)
```

A template approach is to add a properties file in *templates/* directory relative to the ZookeeperService class:

```
dataDir={{ DATA_DIR }}
clientPort=2181
{% for node in nodes %}
server.{{ loop.index }}={{ node.account.hostname }}:2888:3888
{% endfor %}
```

Suppose we named the file zookeeper.properties. The creation of the config file will look like this:

```
prop_file = self.render('zookeeper.properties')
node.account.create_file(self.CONFIG_FILE, prop_file)
```

CHAPTER 6

Debug Tests

The test results go in results/<date>--<test_number>. For results from a particular test, look for results/<date>--<test_number>/test_class_name/<test_method_name>/ directory. The test_log.debug file will contain the log output from the python driver, and logs of services used in the test will be in service_name/node_name sub-directory.

If there is not enough information in the logs, you can re-run the test with --no-teardown argument.

```
ducktape dir/tests/my_test.py::TestA.test_a --no-teardown
```

This will run the test but will not kill any running processes or remove log files when the test finishes running. Then, you can examine the state of a running service or the machine when the service process is running by logging into that machine. Suppose you suspect a particular service being the cause of the test failure. You can find out which machine was allocated to that service by either looking at test_log.debug or at directory names under results/<date>--<test_number>/test_class_name/<test_method_name>/service_name/. It could be useful to add an explicit debug log to start_node method with a node ID and node's hostname information for easy debugging:

```
def start_node(self, node):
    idx = self.idx(node)
    self.logger.info("Starting ZK node %d on %s", idx, node.account.hostname)
```

The log statement will look something like this:

```
[INFO  - 2017-03-28 22:07:25,222 - zookeeper - start_node - lineno:50]: Starting ZK␣
↪node 1 on worker1
```

If you are using Vagrant for example, you can then log into that node via:

```
vagrant ssh worker1
```

6.1 Use Logging

Distributed system tests can be difficult to debug. You want to add a lot of logging for debugging and tracking progress of the test. A good approach would be to log an intention of an operation with some useful information before any operation that can fail. It could be a good idea to use a higher logging level than you would in production so more info is available. For example, make your log levels default to DEBUG instead of INFO. Also, put enough information to a message of `assert` to help figure out what went wrong as well as log messages. Consider an example of testing ElasticSearch service:

```
res = es.search(index="test-index", body={"query": {"match_all": {}}})
self.logger.debug("result: %s" % res['hits'])
assert res['hits']['total'] == 1, "Expected total 1 hit, but got %d" % res['hits'][
↪'total']
for hit in res['hits']['hits']:
    assert 'kimchy’ == hit['_source']['author’], "Expected author kimchy but got %s"
↪% hit['_source']['author']
    assert 'Elasticsearch: cool.' == hit['_source']['text’], "Expected text␣
↪Elasticsearch: cool. but got %s" % hit['_source']['text’]
```

First, the tests outputs the result of a search, so that if any of the following assertions fail, we can see the whole result in `test_log.debug`. Assertion messages help to quickly see the difference in expected and retrieved results.

6.2 Fail early

Try to avoid a situation where a test fails because of an uncaught failure earlier in the test. Suppose we write a `start_node` method that does not check if the service starts successfully. The service fails to start, but we get a test failure indication that there was a problem querying the service. It would be much faster to debug the issue if the test failure pointed to the issue with starting the service. So make sure to add checks for operations that may fail, and fail the test earlier than later.

6.3 Flaky tests

Flaky tests are hard to debug due to their non-determinism, they waste time, and sometimes hide real bugs: developers tend to ignore those failures, and thus could miss real bugs. Flakiness can come from the test itself, the system it is testing, or the environmental issues.

6.3.1 Waiting on Conditions

A common cause of a flaky test is asynchronous wait on conditions. A test makes an asynchronous call and does not properly wait for the result of the call to become available before using it:

```
node.account.kill_process("zookeeper", allow_fail=False)
time.sleep(2)
assert not self.alive(node), “Expected Zookeeper service to stop”
```

In this example, the test terminates a zookeeper service via `kill_process` and then uses `time.sleep` to wait for it to stop. If terminating the process takes longer, the test will fail. The test may intermittently fail based on how fast a process terminates. Of course, there should be a timeout for termination to ensure that test does not run indefinitely. You could increase sleep time, but that also increases the test run length. A more explicit way to express this condition is to use `wait_until()` with a timeout:

```
node.account.kill_process("zookeeper", allow_fail=False)
wait_until(lambda: not self.alive(node),
           timeout_sec=5,
           err_msg="Timed out waiting for zookeeper to stop.")
```

The test will progress as soon as condition is met, and timeout ensures that the test does not run indefinitely if termination never ends.

Think carefully about the condition to check. A common source of issues is incorrect choice of condition of successful service start in `start_node` implementation. One way to check that a service starts successfully is to wait for some specific log output. However, make sure that this specific log message is always printed after the things run successfully. If there is still a chance that service may fail to start after the log is printed, this may cause race conditions and flaky tests. Sometimes it could be better to check if the service runs successfully by querying a service or checking some metrics if they are available.

6.3.2 Test Order Dependency

Make sure that your services properly cleanup the state in `clean_node` implementation. Failure to properly clean up the state can cause the next run of the test to fail or fail intermittently if other tests happen to clean same directories for example. One of the benefits of isolation that ducktape assumes is that you can assume you have complete control of the machine. It is ok to delete the entire working space. It is also safe to kill all java processes you can find rather than being more targeted. So, clean up aggressively.

6.3.3 Incorrect Assumptions

It is possible that assumptions about how the system works that we are testing are incorrect. One way to help debug this is to use more detailed comments why certain checks are made.

6.4 Tools for Managing Logs

Analyzing and matching up logs from a distributed service could be time consuming. There are many good tools for working with logs. Examples include http://lnav.org/, http://list.xmodulo.com/multitail.html, and http://glogg.bonnefon.org/.

6.5 Validating Ssh Issues

Ducktape supports running custom validators when an ssh error occurs, allowing you to run your own validation against a host. this is done simply by running ducktape with the *–ssh-checker-function*, followed by the module path to your function, so for instance:

```
ducktape my-test.py --ssh-checker-function my.module.validator.validate_ssh
```

this function will take in the ssh error raised as its first argument, and the remote account object as its second.

CHAPTER 7

API Doc

7.1 Test

class `ducktape.tests.test.`**`Test`**(*test_context*, **args*, ***kwargs*)

Bases: `ducktape.template.TemplateRenderer`

Base class for tests.

`__init__`(*test_context*, **args*, ***kwargs*)

`compress_service_logs`(*node*, *service*, *node_logs*)

Compress logs on a node corresponding to the given service.

Parameters

- **`node`** – The node on which to compress the given logs
- **`service`** – The service to which the node belongs
- **`node_logs`** – Paths to logs (or log directories) which will be compressed

Returns a list of paths to compressed logs.

`copy_service_logs`(*test_status*)

Copy logs from service nodes to the results directory.

If the test passed, only the default set will be collected. If the the test failed, all logs will be collected.

`min_cluster_size`()

Returns the number of linux nodes which this test needs.

THIS METHOD IS DEPRECATED, and provided only for backwards compatibility. Please implement min_cluster_spec instead.

Returns An integer.

min_cluster_spec()
Returns a specification for the minimal cluster we need to run this test.

This method replaces the deprecated min_cluster_size. Unlike min_cluster_size, it can handle non-Linux operating systems.

In general, most Tests don't need to override this method. The default implementation seen here works well in most cases. However, the default implementation only takes into account the services that exist at the time of the call. You may need to override this method if you add new services during the course of your test.

Returns A ClusterSpec object.

setup()
Override this for custom setup logic.

teardown()
Override this for custom teardown logic.

class ducktape.tests.test.**TestContext**(***kwargs*)
Bases: `object`

Wrapper class for state variables needed to properly run a single 'test unit'.

__init__(***kwargs*)

Parameters

- **session_context** –
- **cluster** – the cluster object which will be used by this test
- **module** – name of the module containing the test class/method
- **cls** – class object containing the test method
- **function** – the test method
- **file** – file containing this module
- **injected_args** – a dict containing keyword args which will be passed to the test method
- **cluster_use_metadata** – dict containing information about how this test will use cluster resources

close()
Release resources, etc.

copy(***kwargs*)
Construct a new TestContext object from another TestContext object Note that this is not a true copy, since a fresh ServiceRegistry instance will be created.

description
Description of the test, needed in particular for reporting. If the function has a docstring, return that, otherwise return the class docstring or "".

expected_cluster_spec
The cluster spec we expect this test to consume when run.

Returns A ClusterSpec object.

expected_num_nodes
How many nodes of any type we expect this test to consume when run.

Returns an integer number of nodes.

local_scratch_dir
This local scratch directory is created/destroyed on the test driver before/after each test is run.

test_name
The fully-qualified name of the test. This is similar to test_id, but does not include the session ID. It includes the module, class, and method name.

7.2 Services

class `ducktape.services.service.`**Service**(*context*, *num_nodes=None*, *cluster_spec=None*, **args*, ***kwargs*)

Bases: `ducktape.template.TemplateRenderer`

Service classes know how to deploy a service onto a set of nodes and then clean up after themselves.

They request the necessary resources from the cluster, configure each node, and bring up/tear down the service.

They also expose information about the service so that other services or test scripts can easily be configured to work with them. Finally, they may be able to collect and check logs/output from the service, which can be helpful in writing tests or benchmarks.

Services should generally be written to support an arbitrary number of nodes, even if instances are independent of each other. They should be able to assume that there won't be resource conflicts: the cluster tests are being run on should be large enough to use one instance per service instance.

__init__(*context*, *num_nodes=None*, *cluster_spec=None*, **args*, ***kwargs*)
Initialize the Service.

Note: only one of (num_nodes, cluster_spec) may be set.

Parameters

- **context** – An object which has at minimum 'cluster' and 'logger' attributes. In tests, this is always a TestContext object.
- **num_nodes** – An integer representing the number of Linux nodes to allocate.
- **cluster_spec** – A ClusterSpec object representing the minimum cluster specification needed.

allocate_nodes()
Request resources from the cluster.

allocated
Return True iff nodes have been allocated to this service instance.

clean()
Clean up persistent state on each node - e.g. logs, config files etc. Subclasses must override clean_node.

clean_node(*node*)
Clean up persistent state on this node - e.g. service logs, configuration files etc.

close()
Release resources.

cluster
The cluster object from which this service instance gets its nodes.

free()
Free each node. This 'deallocates' the nodes so the cluster can assign them to other services.

get_node(*idx*)
ids presented externally are indexed from 1, so we provide a helper method to avoid confusion.

idx(*node*)
Return id of the given node. Return -1 if node does not belong to this service.

idx identifies the node within this service instance (not globally).

local_scratch_dir
This local scratch directory is created/destroyed on the test driver before/after each test is run.

logger
The logger instance for this service.

run()
Helper that executes run(), wait(), and stop() in sequence.

static run_parallel(**args*)
Helper to run a set of services in parallel. This is useful if you want multiple services of different types to run concurrently, e.g. a producer + consumer pair.

service_id
Human-readable identifier (almost certainly) unique within a test run.

start()
Start the service on all nodes.

start_node(*node*)
Start service process(es) on the given node.

stop()
Stop service processes on each node in this service. Subclasses must override stop_node.

stop_node(*node*)
Halt service process(es) on this node.

wait(*timeout_sec=600*)
Wait for the service to finish. This only makes sense for tasks with a fixed amount of work to do. For services that generate output, it is only guaranteed to be available after this call returns.

wait_node(*node*, *timeout_sec=None*)
Wait for the service on the given node to finish. Return True if the node finished shutdown, False otherwise.

who_am_i(*node=None*)
Human-readable identifier useful for log messages.

class ducktape.services.background_thread.**BackgroundThreadService**(*context*, *num_nodes*)
Bases: *ducktape.services.service.Service*

__init__(*context*, *num_nodes*)

wait(*timeout_sec=600*)
Wait no more than timeout_sec for all worker threads to finish.

raise TimeoutException if all worker threads do not finish within timeout_sec

7.3 Remote Account

class `ducktape.cluster.remoteaccount.`**`RemoteAccount`**(*ssh_config*, *externally_routable_ip=None*, *logger=None*, *ssh_exception_checks=[]*)

Bases: `ducktape.utils.http_utils.HttpMixin`

RemoteAccount is the heart of interaction with cluster nodes, and every allocated cluster node has a reference to an instance of RemoteAccount.

It wraps metadata such as ssh configs, and provides methods for file system manipulation and shell commands.

Each operating system has its own RemoteAccount implementation.

`__init__`(*ssh_config*, *externally_routable_ip=None*, *logger=None*, *ssh_exception_checks=[]*)

`alive`(*pid*)
Return True if and only if process with given pid is alive.

`close`()
Close/release any outstanding network connections to remote account.

`copy_between`(*src*, *dest*, *dest_node*)
Copy src to dest on dest_node

Parameters

- **`src`** – Path to the file or directory we want to copy
- **`dest`** – The destination path
- **`dest_node`** – The node to which we want to copy the file/directory

Note that if src is a directory, this will automatically copy recursively.

`java_pids`(*match*)
Get all the Java process IDs matching 'match'.

Parameters **`match`** – The AWK expression to match

`kill_java_processes`(*match*, *clean_shutdown=True*, *allow_fail=False*)
Kill all the java processes matching 'match'.

Parameters

- **`match`** – The AWK expression to match
- **`clean_shutdown`** – True if we should shut down cleanly with SIGTERM; false if we should shut down with SIGKILL.
- **`allow_fail`** – True if we should throw exceptions if the ssh commands fail.

`monitor_log`(*log*)
Context manager that returns an object that helps you wait for events to occur in a log. This checks the size of the log at the beginning of the block and makes a helper object available with convenience methods for checking or waiting for a pattern to appear in the log. This will commonly be used to start a process, then wait for a log message indicating the process is in a ready state.

See `LogMonitor` for more usage information.

`remove`(*path*, *allow_fail=False*)
Remove the given file or directory

wait_for_http_service(*port*, *headers*, *timeout=20*, *path='/'*)
Wait until this service node is available/awake.

class ducktape.cluster.remoteaccount.**LogMonitor**(*acct*, *log*, *offset*)
Bases: object

Helper class returned by monitor_log. Should be used as:

```
with remote_account.monitor_log("/path/to/log") as monitor:
    remote_account.ssh("/command/to/start")
    monitor.wait_until("pattern.*to.*grep.*for", timeout_sec=5)
```

to run the command and then wait for the pattern to appear in the log.

__init__(*acct*, *log*, *offset*)

wait_until(*pattern*, ***kwargs*)
Wait until the specified pattern is found in the log, after the initial offset recorded when the LogMonitor was created. Additional keyword args are passed directly to ducktape.utils.util.wait_until

class ducktape.cluster.linux_remoteaccount.**LinuxRemoteAccount**(**args*, ***kwargs*)
Bases: *ducktape.cluster.remoteaccount.RemoteAccount*

__init__(**args*, ***kwargs*)

get_external_accessible_network_devices()
gets the subset of devices accessible through an external conenction

get_network_devices()
Utility to get all network devices on a linux account

local
Returns True if this 'remote' account is probably local. This is an imperfect heuristic, but should work for simple local testing.

class ducktape.cluster.windows_remoteaccount.**WindowsRemoteAccount**(**args*, ***kwargs*)
Bases: *ducktape.cluster.remoteaccount.RemoteAccount*

Windows remote accounts are currently only supported in EC2. See _setup_winrm() for how the WinRM password is fetched, which is currently specific to AWS.

The Windows AMI needs to also have an SSH server running to support SSH commands, SCP, and rsync.

__init__(**args*, ***kwargs*)

7.4 Clusters

class ducktape.cluster.cluster.**Cluster**
Bases: object

Interface for a cluster – a collection of nodes with login credentials. This interface doesn't define any mapping of roles/services to nodes. It only interacts with some underlying system that can describe available resources and mediates reservations of those resources.

__init__()

all()
Return a ClusterSpec object describing all nodes.

alloc(*cluster_spec*)
Allocate some nodes.

Parameters **`cluster_spec`** – A ClusterSpec describing the nodes to be allocated.

Throws InsufficientResources If the nodes cannot be allocated.

Returns Allocated nodes spec

`available`()
Return a ClusterSpec object describing the currently available nodes.

`do_alloc`(*cluster_spec*)
Subclasses should implement actual allocation here.

Parameters **`cluster_spec`** – A ClusterSpec describing the nodes to be allocated.

Throws InsufficientResources If the nodes cannot be allocated.

Returns Allocated nodes spec

`free`(*nodes*)
Free the given node or list of nodes

`used`()
Return a ClusterSpec object describing the currently in use nodes.

class `ducktape.cluster.vagrant.`**`VagrantCluster`**(**args*, ***kwargs*)
Bases: *`ducktape.cluster.json.JsonCluster`*

An implementation of Cluster that uses a set of VMs created by Vagrant. Because we need hostnames that can be advertised, this assumes that the Vagrant VM's name is a routeable hostname on all the hosts.

- If cluster_file is specified in the constructor's kwargs (i.e. passed via command line argument –cluster-file) - If cluster_file exists on the filesystem, read cluster info from the file - Otherwise, retrieve cluster info via "vagrant ssh-config" from vagrant and write cluster info to cluster_file
- Otherwise, retrieve cluster info via "vagrant ssh-config" from vagrant

`__init__`(**args*, ***kwargs*)

class `ducktape.cluster.localhost.`**`LocalhostCluster`**(**args*, ***kwargs*)
Bases: *`ducktape.cluster.cluster.Cluster`*

A "cluster" that runs entirely on localhost using default credentials. This doesn't require any user configuration and is equivalent to the old defaults in cluster_config.json. There are no constraints on the resources available.

`__init__`(**args*, ***kwargs*)

class `ducktape.cluster.json.`**`JsonCluster`**(*cluster_json=None*, **args*, ***kwargs*)
Bases: *`ducktape.cluster.cluster.Cluster`*

An implementation of Cluster that uses static settings specified in a cluster file or json-serializeable dict

`__init__`(*cluster_json=None*, **args*, ***kwargs*)
Initialize JsonCluster

JsonCluster can be initialized from:

- a json-serializeable dict
- a "cluster_file" containing json

Parameters

- **`cluster_json`** – a json-serializeable dict containing node information. If `cluster_json` is None, load from file
- **`(optional)`** (*`cluster_file`*) – Overrides the default location of the json cluster file

Example json with a local Vagrant cluster:

```
{
  "nodes": [
    {
      "externally_routable_ip": "192.168.50.151",

      "ssh_config": {
        "host": "worker1",
        "hostname": "127.0.0.1",
        "identityfile": "/path/to/private_key",
        "password": null,
        "port": 2222,
        "user": "vagrant"
      }
    },
    {
      "externally_routable_ip": "192.168.50.151",

      "ssh_config": {
        "host": "worker2",
        "hostname": "127.0.0.1",
        "identityfile": "/path/to/private_key",
        "password": null,
        "port": 2223,
        "user": "vagrant"
      }
    }
  ]
}
```

static make_remote_account(*ssh_config*, **args*, ***kwargs*)
Factory function for creating the correct RemoteAccount implementation.

7.5 Template

class ducktape.template.**TemplateRenderer**
Bases: `object`

render(*path*, ***kwargs*)
Render a template loaded from a file. template files referenced in file f should be in a sibling directory of f called "templates".

Parameters

- **path** – path, relative to the search paths, to the template file
- **kwargs** – optional override parameters

Returns the rendered template

render_template(*template*, ***kwargs*)
Render a template using the context of the current object, optionally with overrides.

Parameters

- **template** – the template to render, a Template or a str
- **kwargs** – optional override parameters

Returns the rendered template

CHAPTER 8

Misc

8.1 Developer Install

If you are are a ducktape developer, consider using the develop command instead of install. This allows you to make code changes without constantly reinstalling ducktape (see http://stackoverflow.com/questions/19048732/python-setup-py-develop-vs-install for more information):

```
cd ducktape
python setup.py develop
```

To uninstall:

```
cd ducktape
python setup.py develop --uninstall
```

8.2 Unit Tests

You can run the tests with code coverage and style check using tox:

```
tox
```

Alternatively, you can activate the virtualenv and run pytest and flake8 directly:

```
source ~/.virtualenvs/ducktape/bin/activate
pytest tests
flake8
```

8.3 Windows

Ducktape support Services that run on Windows, but only in EC2.

When a `Service` requires a Windows machine, AWS credentials must be configured on the machine running ducktape.

Ducktape uses the boto3 Python module to connect to AWS. And `boto3` support many different configuration options

Here's an example bare minimum configuration using environment variables:

```
export AWS_ACCESS_KEY_ID="ABC123"
export AWS_SECRET_ACCESS_KEY="secret"
export AWS_DEFAULT_REGION="us-east-1"
```

The region can be any AWS region, not just `us-east-1`.

CHAPTER 9

Contribute

- Source Code: https://github.com/confluentinc/ducktape
- Issue Tracker: https://github.com/confluentinc/ducktape/issues

CHAPTER 10

License

The project is licensed under the Apache 2 license.

Index