Madison Test Suite - A Portable and Extensible Test Framework for Linux Clusters

Kalyana Chadalavada, Shrankhla Upadhyay
HPCC, Enterprise Solutions Engineering,
Bangalore Development Center, Dell Inc.
{kalyana_chadalavada, shrankhla_upadhyay}@dell.com

Abstract:

Clustering has become the architecture of choice for the new generation supercomputing systems. A lot of cluster management suites started making inroads in to the datacenters to meet this requirement. These management software are intelligent enough to detect and identify all the layers of a cluster and usually provide a single point of control for the system administrators. This brings in an entirely new challenge to the developers - verification of their software on a cluster that, theoretically, has no bounds. The developers have to make sure that the various modules that make up the software are working as designed across a large number of nodes. It would be quite tedious if one had to verify the correct functioning of all the components across the cluster.

A comprehensive test suite should be able to test all of these components with user dependency. The suite should leverage existing information, execute the requested test cases and report the results in a structured fashion. It should also provide room for expandability so new test cases and system parameters can be added on demand. We propose a test-suite which sits on an expandable and portable architecture, and comes with a host of features that make it very easy for even a layman to verify various functionalities across a cluster.

Need for a cluster management software test framework:

A cluster management software suite involves a set of components that work very inter-dependently to proactively monitor and maintain the cluster at an optimal state. These components are often developed by independent teams and are put through a set of quality assurance tests. They are tested independently and then integrated in to a single suite. The challenge lies in testing these components as a single unit across a cluster keeping in consideration the target size of the deployment scenario. The following are some challenges that need to be addressed:

- Time consuming to test all components individually on all nodes or a set of nodes selected via random sampling.
- High investment costs involved in terms of human resources required driving down profit margins
- Loss of competitive advantage in terms of time to market

Hence there is a need for an automated mechanism to test all the components in the cluster management suite in a time and resource efficient manner. In addition to addressing the above challenges, the proposed mechanism should also be extensible and portable across platforms.

Proposed Solution Architecture:

Clusters typically consist of one or more "master" nodes that act as the single point of control for the entire cluster. This master node utilizes the individual software components to gather data about the various cluster components. It also maintains information about the components themselves like the IP address of the component, what role is the component currently playing in the cluster and the like. This information can be queried using provided command line interfaces or directly connecting to the data source used by the master node.

We propose the following architecture in order to address the challenge of testing a cluster management suite efficiently in terms of resources and time thereby increasing time to market and driving down costs. [Fig. 1]

There are seven major components of the framework. They are:

1. Driver
2. Host Manager
3. Runtime Environment Data Store
4. Test Manager
5. Utilities
6. Various Configuration Files
7. Test cases

1. Driver:

The driver is the component that co-ordinates the execution of the suite. This is the interface for the user through which the desired environment and test cases are submitted to the suite. The driver utilizes other modules to read, organize and process the given data. This data is then passed on to the appropriate modules to maintain a global state and execute various test cases. The driver should accept the preferred devices and format of the output. It will also structure the output provided by the test cases.

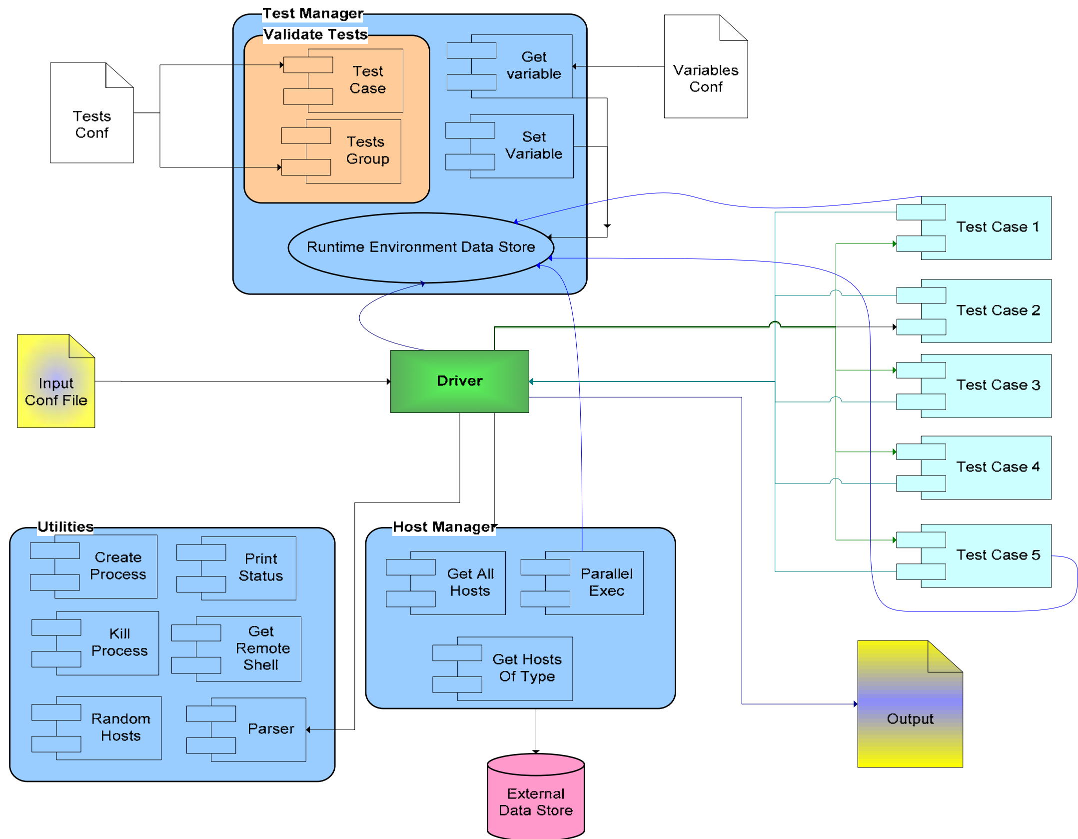

Fig.1 Test Framework Architecture

2. Host Manager:

This component hosts all the functions that are needed to execute various operations on the available set of hosts. It will mainly enable the requestor to obtain the set of hosts available, set of hosts playing a particular role and to execute a command across a set of given hosts. Any test case or module that wishes to execute a set of commands across a set of nodes can utilize this module for the same without worrying about the specifics of implementation.

3. Runtime Environment Data Store:

Runtime Environment Data Store is a protected object maintained during the execution of the test suite. This object maintains a current list of valid variables and their values. This object is initialized during the startup of the suite with a set of default values maintained in a configuration file. The values provided by the user are then super-imposed over this object to overwrite the values of *only* the desired variables while maintaining the default values of other variables. This object can be queried through a set of routines to obtain the values for any valid variable. Similarly, the driver, parser and other special routines can set the value of a variable or reset the value(s) of a particular variable or all variables to default value(s). New <name,

value> pairs can be added to this object during runtime. This process will ensure that all the test cases are operating with a set of consistent values through out the execution cycle.

4. Test Manager:

Test manager hosts a set of routines that help in maintaining a consistent environment during the execution of the test cases. It mainly hosts the Runtime Environment Data Store (described above) and another interface for test validation. The test validation interface initializes itself with a list of valid test case names and valid group names from a configuration file. This interface accepts a user friendly representation of comma separated list of test cases requested as provided by the parser and returns a well formed data structure containing a list of valid and invalid test cases and group names. The driver can use this well defined data structure to invoke various test case execution modules. User can create his own groups from a set of valid test case names. Recursion of groups is left to the implementation.

5. Utilities:

As the name suggests, utilities will be a collection of helper functions of the test suite. This will primarily include a parser, printer, process management functions and a routine to detect and set the remote shell utility.

The parser will read the user provided configuration file and return a set of <name, value> pairs to the driver. The driver will process these <name, value> pairs by identifying the type of the variable by its name. It may send the pair off to the validateTest module to obtain a list of valid tests or to the host manager to set the hosts to be used during the current execution.

The printer will help print status messages and output in a consistent manner to the device of choice.

Process management functions will spawn and monitor a set of child processes as requested by any test case implementations. The module should be able to successfully spawn, monitor and terminate jobs on demand.

GetRemoteShell will enable to suite to determine the preferred utility for executing commands on remote machines. This routine must default to "*ssh*" in absence of an explicit setting.

6. Various configuration files:

There are three important configuration files that are used. All files are optional or can be implemented in other ways unless specified specifically.

a. Input configuration file: This file is provided by the user with his preferences for various variables. The driver accepts this file and passes it on to the parser. In this file is not provided, the test suite can default to a set of basic tests with hostnames pulled in from the external data store maintained by the cluster management suite.
b. Variable configuration file: This file holds the default values for all valid variables. Values are read from this file during initialization and the Runtime Environment Data Store is populated. If this file is missing, it is possible to execute the tests using the values provided by the user.
c. Tests configuration file: This file holds the list of valid test case names and group names. Without this file, it will not be possible for the validateTests module to function properly. This information is mandatory and has to be maintained in a non-volatile medium.
d. External data store: This is a generic term that refers to the data store managed by the cluster management software. The test framework should default to using this as the source for hostnames.

7. Test Cases:

Implementation of the test cases will be totally independent of the test framework. The test case will be invoked by the driver with a list of hosts to be used for the execution of this test case and the output format desired. The test case can query the Runtime Environment Data Store to obtain the list of values it needs for executing the test case. It can utilize the Host Manager module to execute any command across multiple nodes in the cluster. Each test case implementation is expected to format its output and provide it back the driver but should not stream to the standard output. This restriction is implemented because the test case doesn't know if it is executing as a part of a group or just a stand-alone test case. The Driver would structure the output appropriately. A test case implementation must also stream all its status messages to standard error device only. When a new test case is added, the name of the test case needs to be added to the list of valid tests in the tests configuration file. If the test case needs any specific variable that needs to be added to the variable configuration file. The driver should be made aware of this test case by adding the appropriate function call.

Implementation - Madison Test Suite:

With the above architecture, we implemented a test suite, Madison Test Suite, for use with Platform ROCKS [1]. The following are the details of that implementation.

To choose the language for implementation, we considered the following:

a. Portability
b. Availability on a wide range of platforms
c. Ability to parse and modify strings easily
d. Ability to connect to a variety of data sources
e. Minimal learning curve
f. Object oriented programming support

We identified Python [2] as the ideal candidate satisfying all over requirements. Hence we used Python for our implementation.

All the above components were implemented as functions in appropriate Python modules.

1. **Driver:** The driver is the ***only*** script in the implementation. This script will import all the necessary modules and will co-ordinate the transfer of data and state among these various modules.
2. **Host Manager:** This module contains the three specified routines. The parallel execute function has been implemented as a derived class from the NPACI ROCKS [3] implementation of Cluster-Fork [4] utility. The derived class adds support for the ANL's [5] scalable *mpd* process launcher [6] and returns the result with a list of dead nodes in a dictionary data type.
3. **Runtime Environment Data Store:** A Python class with a dictionary was used to implement this component. The class provides a public function for "getVariable" and a protected function for "setVariable".
4. **Test Manager:** This is a python module that hosts the Runtime Environment Data Store object, set of default values for the object and a class for validateData component. This component returns a list of strings with defined delimiters for identifying the beginning and ending of groups. Multiple group recursions are allowed.
5. **Utilities:** This component also was implemented as a Python module. The parser returns a list of dictionaries. Each dictionary contains a set of variables provided and a list of tests. This will enable the user to execute the same set of tests with multiple values for the same variables one after another using a single input file. This is very useful when executing the tests over a set of values. The tests can be submitted and left to complete. A single tester will be able to handle multiple clusters without much effort. The printer is implemented using standard formatting functions. getRemoteShell gets the value from an environment variable.
6. **Various Configuration Files:** We used a simple "*variable = value*" format for all our configuration files with a leading # signifying a comment. Python's split routine was used to separate the variable and value in to two different variables

without much effort. The test suite detects if the current environment is based on NPACI ROCKS and uses the cluster database to obtain node names and roles. User can over ride these values via the input configuration file.

7. **Test Cases:** All test cases have been implemented as standard Python modules. All the test cases are invoked only with a list of hostnames that are to be used to execute this test case. We currently have more than forty test cases implemented, some of which are:

    Various compilers, MPI for Ethernet, Myrinet and Infiniband, Ganglia, Platform's Lava job scheduler, Dell utility partition, a set of test for IBRIX Fusion file system, node connectivity and Parallel Virtual File System (PVFS).

Using this implementation, the time to test each release of Platform ROCKS has reduced significantly.

Conclusion:

We proposed architecture for implementing portable and extensible test framework for use on large clusters. We presented a reference implementation of the proposed architecture. We would like to make the following enhancements moving forward:

- Support XML for input and output with a GUI
- Extend to support NPACI ROCKS 4.0 architecture
- Allow for serialization of the current state to support reboot of the frontend as a part of a test case.
- Add more test cases to completely automate testing of Platform ROCKS.

References:

[1]. http://www.platform.com/rocks
[2]. http://www.python.org
[3]. http://www.rocksclusters.org
[4]. http://www.rocksclusters.org/rocks-documentation/3.2.0/launching-interactive-jobs.html#CLUSTER-FORK
[5]. http://www.anl.gov/
[6]. http://www-unix.mcs.anl.gov/~rbutler/mpd/