http://goo.gl/mCtwU

Test Framework Redesign

Version 1.0
2012-11-2
nathan_rutman@xyratex.com

Revision History

Version	Date	Author	Comment
0.1	2012-11-02	Nathan Rutman	initial cut
1.0	2012-11-09	Nathan Rutman	public release

Table of Contents

Introduction

With each new release of Lustre, large-scale users feel compelled to make a tradeoff: whether to take the latest release with the latest features with the knowledge that this is high-risk for bugs, or stick with the older release that has been “stable enough” for their situation.
Part of the problem is the willingness of scale users to “test” a new release. Scale testing has a number of difficulties that preclude quick and easy testing.

Acknowledgements

This proposal is based on ideas first presented in the Lustre TWG mailing list, and further developed for our LAD'12 presentation. Many thanks especially to Roman Grigoryev of Xyratex and Chris Gearing from Intel for many of these ideas.

Problem Statement

By reworking the Lustre testing framework and individual tests to produce higher-quality, more robust, well-controlled, and safer tests, we may improve various aspects of testing Lustre software:

- Inter-version testing. Client software may be older than servers; tests should still be consistent. Today, tests depend on Lustre version, and the framework is therefore not consistent between nodes when running inter-version testing. An independent test RPM can resolve this.
- Directed testing. Developers and users should be able to run directed tests covering specific areas of concern.
- Arbitrary-order testing. Adding randomized and repeated testing can provide a more challenging test environment with improved code coverage.
- Probability-based testing. For some tests (e.g. performance), pass/failure are not simple thresholds but nonetheless may or may not imply a problem. (See Gearing LAD'12).
- Coverage completeness. There are large areas of code that are not currently tested.
- Coverage efficiency. There are many areas of code that are tested repeatedly.
- Human and machine-readable output. Currently test output is human readable, but not easily machine-readable, hindering automation.
- Configuration dependence. Today test success depends heavily on system configuration. The configuration control is inconsistent and potentially dangerous.
- Test safety. Today some tests erase files, directories, fill disks, erase disks, etc. with no warning or recourse.
- Framework self-test. Changes in the framework and tests may cause failures independent of Lustre. The framework should be able to be tested for correctness independently from Lustre as well.
- Extensibility. The framework must be easy to extend, while at the same time the ability to add functionality directly within the tests themselves should be restricted. In this way, the framework will enforce the use of the API via a consistent and controlled test library.
- Automation. Tests should be easily run via an assortment of automated systems, and the results easily machine-parsed.
- Scalability. The current framework is not scalable to large numbers of clients (utilization) or large numbers of servers. Almost all tests today are single-client tests.

Improvements in these areas will make testing on large systems easier, and therefore hopefully more common.
Furthermore, addressing some of these problems will be difficult to accomplish using the current framework language.

Solution Components

First we will describe all the capabilities of an ideal Lustre test framework. In a following section we will describe how to transition incrementally to such an ideal system.

Independent test RPM

Separate the embedded Lustre tests (test-framework.sh, acceptance-small.sh, etc) from the Lustre RPMs, and move them into a dedicated lustre-test.rpm. This allows work on the test rpm to be conducted independently from Lustre releases. Additionally, it simplifies inter-version Lustre testing since there will be a single-version test infrastructure between any Lustre components to be tested. New unit tests developed to exercise new features will correctly fail on versions that do not support these features. This is expected and desired behavior.

Separate configuration from testing

The current test-framework.sh must be explicitly given a configuration description (lustre/tests/cfg dir or environemental vars). This is then used (inconsistently) to format and setup the test filesystem (sometimes), or as a description of the existing filesystem (sometimes). The setup capabilities are not comprehensive (no MDRAID, no VM's, etc). The description may not match the actual existing system. The lack of consistency and clarity here has been an ongoing source of test-debugging problems.
To alleviate these problems, this sometimes-automated configuration and setup should be removed from the testing framework. Testing should always be executed against an existing, configured, working filesystem, given as a parameter to the framework (e.g. mgsnode, fsname). The framework should learn (and record) all of the necessary configuration information from there.
While the capabilities to do some configuration operations will need to be included in the test framework in order to execute particular tests, this requirement should not be overloaded with the requirement to set up the initial filesystem. Tests should never change the configuration unless specifically testing those capabilities, in which case they will need to be given explicit permission, see below, and they will need to restore the original configuration when the test is completed.
For automated testing environments where the filesystems are not permanent, an external mechanism must be used to configure and format the filesystem. Several other more complete automated setup systems currently exist that can be used for this purpose (lustre/scripts for one). Xyratex is also developing an open-source automated test system for Lustre (Xperior) that could also be used for this purpose.

Better test and infrastructure language

Bash, while powerful and portable, does not enforce clean function encapsulation nor provide

easy handling of error cases. Furthermore, parsing of results (especially from remote nodes) tends to be complicated and error-prone. It is not fast, nor does it have a direct Posix API (i.e. no “open”). There are no libraries for remote node control. For these reasons, a more appropriate language should be chosen for a framework redesign.
Ideal language requirements:

- strict function interfaces
- universally available/portable
- widely maintained
- widely understood
- fully-featured filesystem interface: posix API
- fast - replace e.g. createmany with embedded function
- operate remote instances
- inter-version compatibility

Two languages seem to best fit these requirements: Perl and Python. While each have their own proponents, we suggest Python is ultimately the better choice for test clarity.

Enforce test independence

Current tests are labelled inconsistently; sometimes subtest numbering (e.g. 25b) is used for topically related tests, sometimes as separate stages in a single test, and sometimes for no clear reason. Instead, if tests are separate, they should have value when executed separately. Tests must not depend on the success of a previous test in order to run successfully; this can be avoided simply by combining and dependent tests together.

An error function should be called at any stage that may fail, and the test should clean itself up correctly. The error code and test line number of the failure should be reported as the test result.

Every test must restore the original on-disk state of the filesystem. This includes:

- leftover files
- leftover dirs
- parameter settings
- active features (e.g. changelogs, quotas)
- configuration (mounts, active OSTs, etc)
- status (on-line, not readonly, etc)

Meeting these requirements allows the test ordering to be *randomizable* and *repeatable* (multiple runs of the same test). Furthermore, it allows (in some cases) for *parallel* test execution, where different clients may be executing different tests simultaneously. Note that it does not necessarily allow for parallel test execution in all cases, because they may conflict (e.g. set opposite values for a parameter or both use obd_fail_loc).

Add test metadata

Current tests are not self-descriptive: typically one must infer the purpose from the output line:

```
run_test 25b "lookup file in symlinked directory ==============="
```
Instead, metadata should be attached systematically to each test to provide clarity for:

- intent
    - verifies symlinked directories are usable for file lookup
- requirements
    - node access
        - client
    - permissions
        - file, dir creation
        - file, dir removal
- coverage areas
    - POSIX
    - symlink
    - file lookup
- implications of running the test
    - temporary directory, file, symlink is created and removed
- meta
    - old_name: “sanity 25b”

Note that if we implement the API correctly, the “requirements” section should be able to be created automatically based on API calls - system requirements, node access, and framework permissions should be enforced by the API itself.

Permissions mask

For some tests (e.g. conf-sanity), it is impossible to separate the configuration from the tests themselves. Many times, configuration changes are dangerous. This becomes a driver for a *permissions mask* that explicitly grants permission for tests to perform various operations:

- small client-based filesystem operations
- large (OST fill) client ops
- heavy loading (IO or MD ops)
- change debugging levels and capture log output
- set temporary parameters (set_param)
- set permanent parameters (conf_param)
- enable obd_fail_loc
- register a changelog user
- start/stop Lustre clients
- start/stop Lustre servers
- failover node to itself
- failover a node to partner
- add or remove servers
- reformat a target

Every test that performs one of these actions should do so via an API call, which will verify the permissions given to the testing run before executing the operation.

YAML Output

Output from both individual tests and from the framework should be easily understandable to both humans and machines. To this end, we standardize on YAML formats for all test and framework output.
Output should contain at least:

- descriptions of what is being tested
- environment
- completion status
- test results
- timing information
- error codes with test line numbers for failures/exits within a test

See Appendix A for a sample YAML test output.

Add Framework Self-Test

The framework itself should be build with consistency and correctness unit tests, to verify the correct functionality of the test execution, measurement, and reporting functions. These should be executed independently from any Lustre features.

Statistical-Based Testing

Analysis of tests should include concepts for statistical models for what constitutes “success” and “failure” of a test (see Chris Gearing’s LAD presentation). Simple thresholds for performance metrics may not be able to identify problems or may too aggressively report problems where there are none. Example metrics include IO rates, rpc count, test execution time, and even failure frequency.
Determination of these events may be beyond the scope of the test framework, but the framework must provide sufficient detail in machine-readable form for secondary analysis.

Constraint-based testing

To more fully characterize response in unusual circumstances, it would be useful for the framework to be able to control and vary some aspects of the environment. For example, if the framework were able to specify the amount of memory available to the kernel (e.g. when using a virtual machine), then OOM conditions and behavior could be easily tested. Similarly, limiting the number of MDS threads to (num_clients-1), running with a very high network latency, etc. would provide testing of code paths that are not normally tested.

Framework API

The framework should have a clear, enforceable API for allowing interaction with the filesystem. Areas that should be covered in the API should include the following.

Queries for system configuration

Tests may need to use Lustre or environmental information to alter their behavior. For instance, the number of OSTs will impact the object allocation patterns on the MDS. However, we should generally try to limit the amount of information required by the tests - for example, direct knowledge of the mount point is probably not needed if primitives exist for “mount the OSTs” and “create a file on the Lustre fs”.

Role-based operations

The current tests are mostly written as if intended to be run from a client. However, this is not compatible with some testing systems, where the role of the test executor and the role of the client may not coincide - for example some Cray nodes are not amenable to running scripts this way. Operations that must take place on specific nodes should all use a common API format to specify the node or role. This is similar to the “do_node” function in the current test-framework.sh, but it must be extended to explicit instead of implicit client roles.

Remote operations capability

The ability to run commands on remote nodes must be a first-class component of the test framework. Executing of remote commands should be encapsulated with the framework and hidden from the individual tests. A role and a command should be specified, and the framework should use the appropriate methods depending on those available to the environment. Some options might be passwordless ssh, psdh, LNET tunnel, or other mechanisms.

Filtering of tests by metadata labels

Test metadata as discussed above should be usable by the test framework in order to make decisions about the appropriateness of a test. Certain label values may cause a test to be excluded or included from a given test run. For example, if the framework is set to filter on “recovery” tests, only tests that have to do with recovery would be executed. This allows for more focused testing of particular areas of concern for developers. This may potentially provide more in-depth test coverage of a particular area, or reduce the testing time for development of given feature.

Precondition checks

Current conditions of the configuration or filesystem state may exclude tests from executing correctly or usefully. These checks should be performed before trying to start a test, so that the output condition of a test is not “skip” (as is the case today), but rather a note in the test log saying the test was skipped due to failure of precondition X. The preconditions should be specified via a special section before each test, perhaps along with the test metadata.

```
test_meta:
    suite: sanity
```

```
        number: 25b
        intent: verifies symlinked directories are usable for file lookup
        precondition: filesystem_is_up -- common requirements have special fns
        precondition: roles(client) -- we have access/permission for a client
        precondition: “rsync --help | grep -q xattr” -- custom check
    begin...
```

Permissions mask

The permissions mask as discussed above should be used to gate execution of permitted operations. Any operation that requires one of the specified permissions should be sent through an API version of the function that does the permission check. For example,

```
safe_set_param(param, value)
```

would call an API function that checks for set_param permission, and executes the command if successful. It would return a special error code on failure that would indicate test failure due to inadequate testing permissions. (Ideally these could even be checked statically before the test is run.)

File creation

File creation requires a rich API as there are many types of files created or used by tests. Some elements of the file creation API should be:

- no explicit file locations / names. API calls should return a name/file handle rather than be given one.
- no assumptions about the underlying Linux distribution/OS or availability of files/dirs for copying. (E.g. tests today assume /etc is populated and used a source of “random files”.)
- a list of created files is maintained, and all files are removed automatically by the framework after the test returns.

Parameter setting

An important part of setting Lustre parameters is that they are correctly restored upon test completion. To that end, a framework call to safe_set_param should:

- check for parameter setting permissions
- maintain a list of parameter changes
- invoke a parallel execution lock if necessary (below...)
- restore the changed parameters automatically upon test completion

Parallel execution lock

A goal of the framework is to allow tests to be executed in parallel with each other, to enhance the code coverage and testing complexity. However, some Lustre parameters affect only “the next RPC” (obd_fail_loc is a prime culprit). So it will be beneficial to have the ability to stop execution of other parallel tests during specific phases of a given test. It would be up to the test framework to enforce this policy rather than Lustre.

The API should provide for a universal “test execution mutex”, that prevents any other test from starting on any other client, and waits for any currently executing tests to complete. Additionally, the API might provide for more sophisticated named mutexes might be used within a single multiple-client test.

Transition Plan

From a practical standpoint we must move incrementally from our current test system to a more ideal solution.

Separate test RPM

The first step will be to move the Lustre tests and test-framework.sh into a separately built and installed package. Once isolated into a unique testing RPM, the framework can reside in an independent git tree, no longer tied to Lustre releases or development. It can then be developed and worked on independently.
There will be no framework changes for this first step; merely a disentaglement of the test system from the Lustre source code.
A reasonable approach after this separation would be to keep one branch of the test rpm an unmodified “old framework” version, were new functional tests can land as developers add new Lustre features. These could be merged into the new framework branches frequently. This avoids the added burden on developers to learn a new language and test system immediately while they are trying to meet code schedules. Once the new framework is stable, new tests should be written to the new API.

Separate dependent tests

Combine or refactor legacy tests to insure they can be run independently. Initial inspection indicates this will not be a difficult task.

Running legacy tests

For some time, the primary effort will be spent on building the new framework, rather than translating individual tests to use the new API. In the meantime, the existing tests will be run via wrapper functions called by the new framework. These wrappers will:

- run existing tests one at a time
- require “all” permissions
- translate output into a standard YAML form as much as possible

Pilot tests

In order to quickly develop a useful new framework, I propose it is worthwhile to translate all the tests from one entire test script (e.g conf-sanity.sh) to use the new framework API. This will insure that the framework and supporting library are complete enough to be useful, as well as providing a direct comparison case with the legacy version of the tests. Both versions would be run initially to gain confidence in and experience with the new system.

Freeze legacy tests

Once the new framework is tested and stable, the legacy shell scripts should be frozen. No new tests will be added to those scripts. New tests will be written using the new API only, and stored in new files (e.g. lt2_sanity for “Lustre Test v2 Sanity”). Both legacy and new tests will be used together for some time.

Replace legacy library

While new functionality is added to the new framework to accommodate new tests, we will also replace the library functions supplied by test-framework.sh with new lt2 versions. As the new versions are implemented, we may trim down test-framework.sh accordingly, and eventually remove it entirely.

Replace legacy tests

On an ongoing basis we should continue to replace legacy tests with lt2 versions in order to take advantage of the many improvements enumerated above.

Conclusion

While the amount of effort involved in replacing the current test framework is significant, the current limitations are also significant. As Lustre continues to increase scale capability, useful testing becomes increasingly critical to helping determine the level of code quality and identify problems before deployment in more critical environments. Therefore I believe it is imperative to develop a much better test system as rapidly as possible.

Appendix A: Example YAML test output

This output is an example of some of the data that should be collected for each test run. It includes environmental information, the exact command that was executed, the test output, and meta-information on the test execution (e.g. timings). It presents data in a form that is both human-understandable and machine parseable. This can be used as a strawman against which to suggest specific improvements.

```
cmd: SLOW=YES  MDSCOUNT=1  mds1_HOST=mft01  mds_HOST=mft01  OSTCOUNT=2   ost1_HOST=mft01
ost2_HOST=mft01  CLIENTS=mft01cl RCLIENTS=\"\"  ONLY=200d DIR=/mnt/lustre
PDSH=\"/usr/bin/pdsh -R ssh -S -w \" /usr/lib64/lustre/tests/sanity.sh 2>
/tmp/test_stderr.316083.log 1>  /tmp/test_stdout.316083.log
completed: yes
dangerous: no
description: Lustre sanity tests
endtime: 1331509252
endtime_planned: 1331509541
executor: XTest::Executor::LustreTests
exitcode: 0
expected_time: 10
```

```
extoptions:
  branch: first
  buildurl:
'http://10.76.49.90:8080/jenkins/job/Luste_testing_2pc_sl61_sl61patchless/./SUITES=sanity
,label=kvmit-patchless/1/'
  configuration: 2VM_SL61_Pathless
  executiontype: IT
  release: 2.1
  sessionstarttime: 1331493894
fail_reason: ' Pool on /mnt/lustre/d200.pools/dir_tst is , not cea1 '
groupname: sanity
id: 200d
killed: no
log:
  console.client: 200d.console.client.log
  console.server: 200d.console.server.log
  diagnostic.xml.server: 200d.diagnostic.xml.server.log
  messages.client: 200d.messages.client.log
  stderr: 200d.stderr.log
  stdout: 200d.stdout.log
masterclient:
  id: client1
  master: yes
  node: client
  type: client
messages: |
  Cannot copy log file [/tmp/messageslog.1331509224.71077]: 256
  Cannot copy log file [/tmp/xpdiagnostic.1331509259.79727.xml]: 256
reference: http://wiki.lustre.org/index.php/Testing_Lustre_Code
result: 'not ok 1  # Pool on /mnt/lustre/d200.pools/dir_tst is , not cea1 '
result_code: 1
roles: StoreSyslog StoreConsole GetDiagnostics
schema: Xperior1
starttime: 1331509241
status: failed
status_code: 1
tags: functional
timeout: 300
```