LUTF High Level Design

Introduction

Currently there is no dedicated functional test tool in Lustre test suites for LNet testing. Lustre Unit Test Framework (LUTF) fills that gap to provide a means for testing existing LNet features as well as new features that would be added in future. It facilitates an easy way of adding new test cases/scripts to test any new LNet feature.

Objectives

This High Level Design Document describes the current LUTF design, code base, infrastructure requirements for its setup and the new features that can be added on top of the current design.

Reference Documents

Document Link
LNet Unit Test Infrastructure (LUTF) Requirements

Document Structure

This document is made up of the following sections:

- Use Cases
- Design Overview
- Building the LUTF
- LUTF-Autotest Integration
- Infrastructure

Use Cases

The LUTF is meant to cover the following test use cases:

Use Case	Description
Single node configuration	• Exercise the `liblnetconfig` API directly to configure LNet • Exercise the `lnetctl` utility to configure LNet All tests are run on one node.
Multi-node/no File system testing	• Configure one or more nodes • Run lnet_selftest • Ensure traffic conforms to configuration • Repeat the above These tests require node synchronization. For example if a script is configuring node A, node B can not start traffic until node A has finished configuration.
Multi-node/File system testing	• Start file system traffic • Perform some configuration changes which would change LNet behaviour • Ensure that configuration changes are honoured These tests require node synchronization.
Error Injection testing	• Either with file system mounted or not • Inject various types of errors on different nodes on the setup • Monitor statistics to determine how LNet is handling faults These tests require node synchronization.

LNet Operations

1. LNet Configuration steps
    a. Via API directly. LUTF will provide a C/Python API to call the liblnetconfig API
    b. Via lnetctl utility. LUTF will provide a simple wrapper class to call lnetctl.
2. Provisioning/Unprovisioning a File System
    a. LUTF will provide an API to provision a file system.
    b. LUTF will provide an API to clean the clustre and get it in a state ready for next test
        i. LUTF will do this automatically before running a test. It'll ensure that the clustre has no FS mounted and no lustre modules loaded. This way a test starts from a clean slate
        ii. LUTF will provide a way to override this feature
3. Verification
    a. This will be the responsibility of each test
4. Running traffic using selftest
    a. LUTF will provide a wrapper class to run selftest, so that the test writer doesn't need to know about selftest specific scripts.

LUTF Provided Infrastructure

1. Provide a python interface to run scripts
    a. Automatically figure out all the suites
    b. Automatically figure out all the tests in each suite
    c. Provide a method to run a script.

i.
```
# Manually running the lutf
# lutf.sh is a wrapper script to run the lutf. It can be called manually or through Auster.
# Takes the following parameters
#   -c config: configuration file with all the environment variable in the same
#              format as what Auster takes. If not provided it'll assume environment
variables are already set.
#   -s: run in shell mode (IE access to python shell)
#       if not provided then run in daemon mode.
# lutf.sh will have the default communication ports hard coded in the script and will start
the agents and the master
#   >> pdsh -w <hostname> <lutf agent start command>
#   >> <lutf bin> <paramters>
>> ./lutf.sh

# when you enter LUTF python interface. It'll have an lutf library already imported

#  environment for test
lutf.get_environment()

# get connected agents
lutf.get_agents()

#print all available suites
lutf.suites

#print all available scripts in the suite
lutf.suites['suite name'].scripts

# reload the suites and the scripts if it has changed
lutf.suites.reload()

# run a script
lutf.suites['suite name'].scripts['script name'].run()

# reload a script after making changes
lutf.suites['suite name'].scripts['script name'].reload()
```

2. Provision LNet configuration
3. Provision FS configuration
4. When running a test script it always makes sure it cleans the clustre
5. Grab common logs
    a. lctl dk
    b. syslog
    c. crash log

LUTF Design Overview

The LUTF is designed with a Master-Agent approach to test LNet. The Master and Agent LUTF instance uses a telnet python module to communicate with each other and more than one Agent can communicate with single Master instance at the same time. The Master instance controls the execution of the python test scripts to test LNet on Agent instances. It collects the results of all the tests run on Agents and write them to a YAML file. It also controls the synchronization mechanism between test-scripts running on different Agents.

The below diagram shows how LUTF interacts with LNet

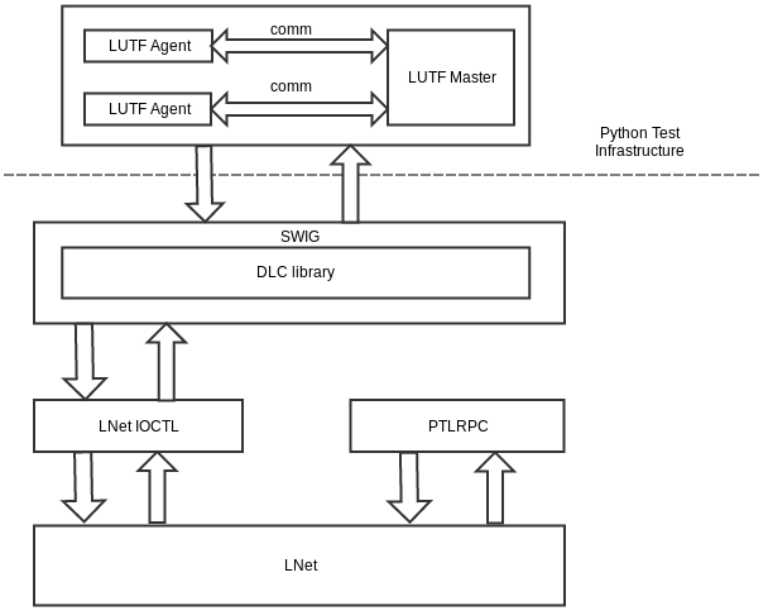
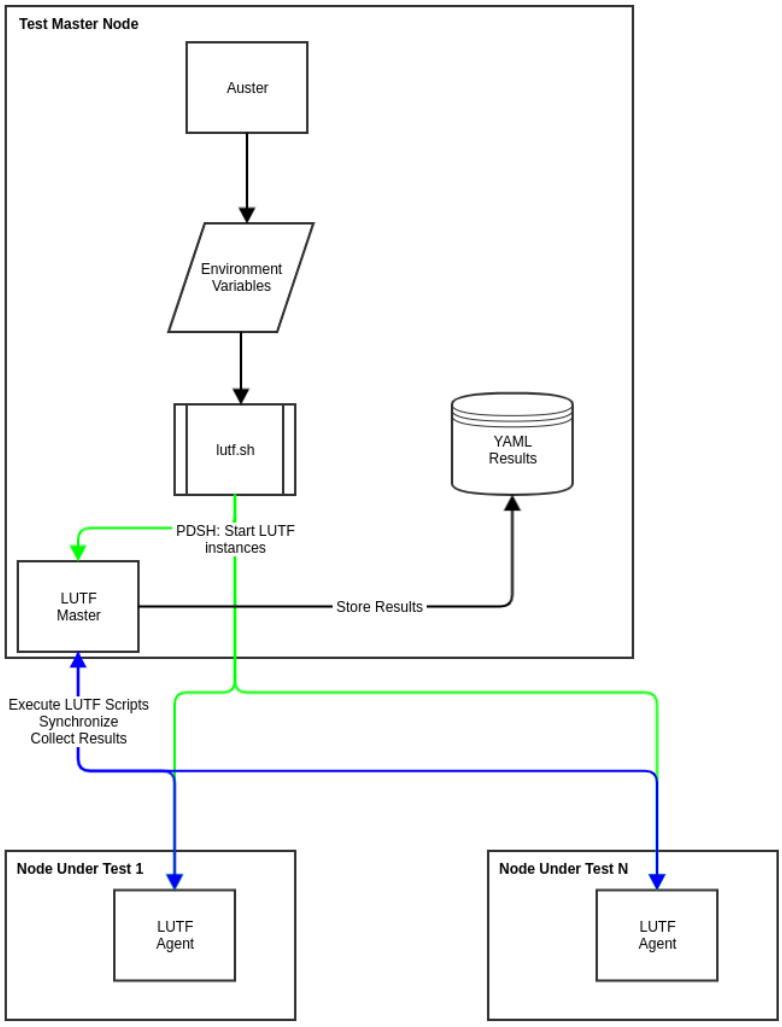
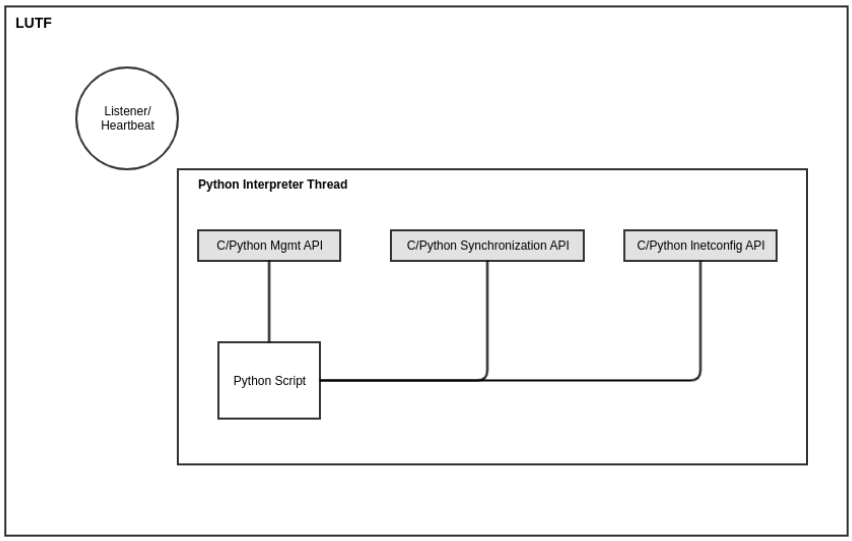


Figure 1: System Level Diagram

LUTF Data Flow



LUTF Threading Overview



The LUTF is designed to allow master-agent, agent-agent or master-master communication. For the first phase of the implementation we will implement the master-agent communication.

Thread Description

- **Listener:** Listens for connections from LUTF Agents and for Heartbeats to monitor aliveness of the Agents.
- **HeartBeat:** Send a periodic heartbeat to the LUTF Master to inform it that the agent is still alive.
- **Python Interpreter:** Executes python test scripts which can call into one of the C/Python APIs provided

C/Python APIs

C/Python Management API

1. Parse configuration
2. provide status on the LUTF Agents
3. provide status on executing scripts
4. Store results

C/Python Synchronization APIs

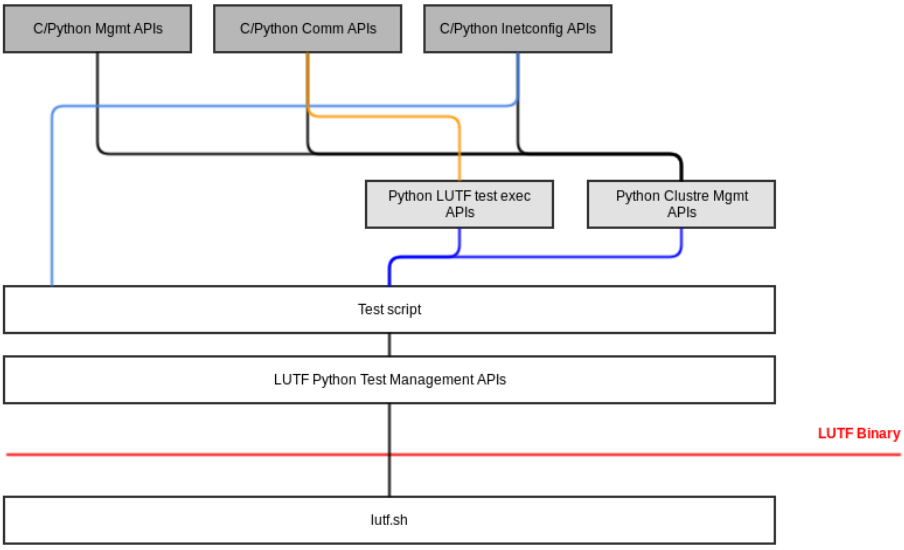
1. Execute tests on the LUTF
    a. This will result in a YAML rpc block being sent to the LUTF agent
2. Wait for work completion events from LUTF Agents
3. Register for asynchronus events
    a. Asynchronous events come in the form of YAML blocks.

C/Python liblnetconfig APIs

1. These are the configuration APIs in `lnet/utils/lnetconfig/liblnetconfig.h`

Other APIs can be wrapped in SWIG and exposed for the LUTF python test scripts to call

LUTF Block view



- C/Python APIs are described above
- Python LUTF test execution APIs: These are a set of classes which allow the abstraction of the execution of python methods on remote nodes
- Python Clustre Managment APIs: These are a set of classes which allow the scripts to manage the cluster: provision it, deploy an LNet configuration, deploy a FS configuration, collect logs, etc.
- Python LUTF test Magamement APIs: These are a set of classes which allow the user to query and execute the LUTF suites and scripts available.
- lutf.sh: A wrapper script which is responsible for starting the LUTF instances on the provisioned clustre

LUTF Deployment

The LUTF will provide a dependency script, `lutf_dep.py`, which will download and install all the necessary elements defined above.

The LUTF will integrate with auster. LUTF should just run like any other Lustre test. A bash wrapper script will be created to execute the LUTF, `lutf.sh` .

SIDE NOTE: Since LUTF simply just runs python scripts, it can run any test, including Lustre tests.

Auster

`auster` configuration scripts set up the environment variables required for the tests to run. These environment variables include:

1. The nodes involved in the tests
2. The devices to use for storage
3. The clients
4. The PDSH command to use

It also sets a host of specific Lustre environment variables.

It then executes the tests scripts, ex: `sanity.sh`

`sanity.sh` can then run scripts utilizing the information provided in the environment variables.

LUTF and Auster

The LUTF will build on the existing test infrastructure.

An `lutf.sh` script will be created, which will be executed from `auster`.

`auster` will continue to setup the environment variables it does as of the time of this writing. The `lutf.sh` will run the LUTF. Since the LUTF is run within the `auster` context, the test python scripts will have access to these environment variables and can use them the same way as the bash test scripts do. If LUTF python scripts are executed on the remote node the necessary information from the environment variables are delivered to these scripts.

Auster will run the LUTF as follows

```
./auster -f lutfcfg -rsv -d /opt/results/ lutf [--suite <test suite name>] [--only <test case name>]
example:
./auster -f lutfcfg -rsv -d /opt/results/ lutf --suite samples --only sample_02
```

Test Prerequisites

Before each test the `lutf.sh` will provide functions to perform the following checks:

1. If the master hasn't started, start it.
2. If the agents on the nodes specified haven't started, then start them.
3. Verify the system is ready to start. IE: master and agents are all started.

Test Post-requisites

1. Provide test results in YAML format.

It's the responsibility of the test scripts to ensure that the system is in an expected state; ie: file system unmounted, modules unloaded, etc.

LUTF Test Scripts Design Overview

- The test scripts will be deployed on all nodes under test as well as the test master.
- Each test script will need to provide a `run` function
    - This function is intended to be executed by the test master
- The LUTF will provide a method to do remote procedure calls.
- Each test, which can be composed of arbitrary python code, must return a YAML text block to the test master reporting the results of the operation.

LUTF Communication Protocol

The Master and the Agent need to exchange information on which scripts to execute and the results of the scripts. Luckily, YAML provides an easy way to transport information. Python YAML parser converts YAML blocks into dictionaries, which are in turn easy to handle in Python code. Therefore YAML is a good way to define Remote Procedure Calls. It is understood that there are other libraries which implement RPCs; however, the intent is to keep the LUTF as simply and easily debug-able as possible.

To execute a function call on a remote node the following RPC YAML block is sent

```
rpc:
   dst: agent_id # name of the agent to execute the function on
   src: source_name # name of the originator of the rpc
   type: function_call # Type of the RPC
   script: script_path # Path to the script which includes the function to execute
   fname: function_name # Name of function to execute
   parameters: # Parameters to pass the function
      param0: value # parameters can be string, integer, float or list
      param1: value2
      paramN: valueN
```

To return the results of the script execution

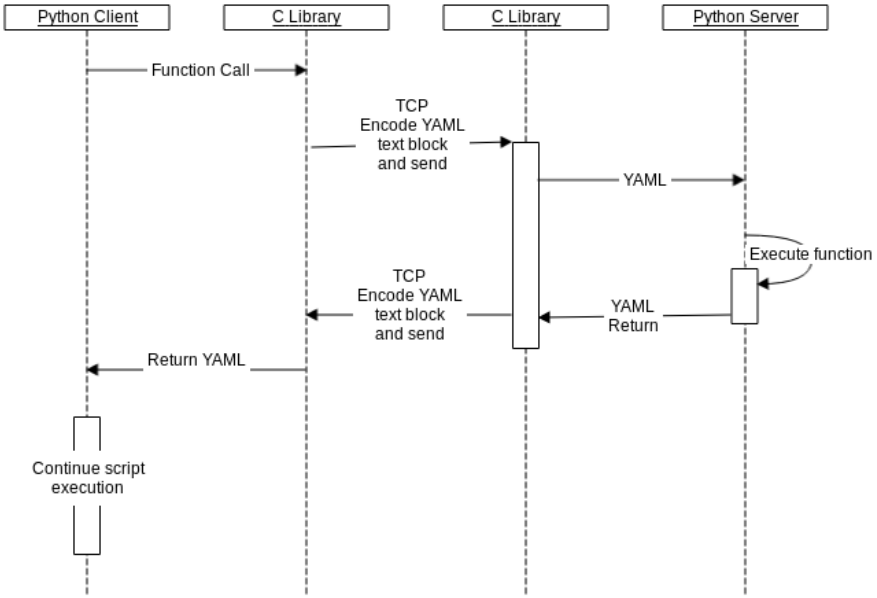
```
rpc:
   dst: agent_id # name of the agent to execute the function on
   src: source_name # name of the originator of the rpc
   type: results # Type of the RPC
   results:
      script: script_path # Path to the script which was executed
      return_code: python_object # return code of function which is a python object
```

A python class will wrap the RPC protocol, such that the scripts do not need to form the RPC YAML block manually.

```
####### Part of the LUTF infrastructure ########
# The BaseTest class is provided by the LUTF infrastructure
# The rpc method of the BaseTest class will take the parameters,
# serialize it into a YAML block and send it to the target specified.
class BaseTest(object, lutfrpc):
   def __init__(target=None):
      if target:
         self.remote = true
         self.target = target

   def __getattribute__(self,name):
        attr = object.__getattribute__(self, name)
        if hasattr(attr, '__call__'):
            def newfunc(*args, **kwargs):
                if self.remote:
                    # execute on the remote defined by:
                    #     self.target
                    #     attr.__name__ = name of function
                    #     type(self).__name__ = name of class
                    result = lutfrpc.send_rpc(self.target, attr.__name__, type(self).__name__, *args, **kwargs)
                else:
                    result = attr(*args, **kwargs)
                return result
            return newfunc
        else:
            return attr

###### In the test script ######
# Each test case will inherit from the BaseTest class.
class Test_1a(BaseTest):
   def __init__(target):
      # call base constructor
      super(Test_1a, self).__init__(target)
   def methodA(parameters):
          # do some test logic
   def methodB(parameters):
      # do some more test logic

# The run function will be executed by the LUTF master
# it will instantiate the Test or the step of the test to run
# then call the class' run function providing it with a dictionary
# of parameters
def run(dictionary, results):
   target = lutf.get_target('mds')
   # do some logic
   Test1a = Test_1a(target);
   result = Test1a.methodA(params)
   if (test for result success):
       result2 = Test1a.methodb(more_params)
   # append the results_yaml to the global results
```

To simplify matters Test parameters take only a dictionary as input. The dictionary can include arbitrary data, which can be encoded in YAML eventually.

Communication Infrastructure



The LUTF provided rpc communciation relies on a simple socket implementation.

1. The LUTF Python RPC call will package the following into a YAML block:
    a. absolute file path
    b. class name
    c. function name
    d. arguments passed to the function
2. The LUTF Python RPC call will call into an LUTF provided C API to send the rpc text block to the target specified and block for response
3. The LUTF slave listener will recieve the rpc YAML text block and pass it up to the python layer
4. Python layer will parse the rpc YAML text block into a python dictionary and will instantiate the class specified and call the method
5. It'll take the return values from the executed method pack it up in an RPC YAML block and call the same C API to send back the YAML block to the waiting master.
6. The master will receive the RPC YAML text block and pass it up to the python RPC layer
7. Python RPC layer will decode the YAML text block into a python dictionary and return the results

This mechanism will also allow the test class methods to be executed locally, by not providing a target

The LUTF can read all the environment variables provided and encode them into the YAML being sent to the node under test. This way the node under test has all the information it needs to execute.

Test Environment Set-Up

Each node which will run the LUTF will need to have the following installed

1. ncurses library
    a. `yum install ncurses-devel`

2. readline library
    a. `yum install readline-devel`

3. rlwrap: Used when telneting into the LUTF telnet server. Allows using up/down errors and other readline features
    a. `yum install rlwrap`
4. python 3.6+
    a. `yum install python3`

5. paramiko
    a. `pip3 install paramiko`
6. netifaces
    a. `pip3 install netifaces`
7. Install PyYAML
    a. `pip3 install pyyaml`

The LUTF will also require that passwordless ssh is setup for all the nodes which run the LUTF. This task is already done when the AT sets up the test cluster.

Building the LUTF

The LUTF shall be integrated with the Lustre tests under `lustre/tests/lutf`. The LUTF will be built and packaged with the standard

```
sh ./autogen.sh
./configure --with-linux=<kernel path>
make
# optionally
make rpms
# optionally
make install
```

The make system will build the following items:

1. `lutf` binary
2. `liblutf_agent.so` - shared library to communicate with the LUTF backend.
3. `clutf_agent.py` and `_clutf_agent.so`: glue code that allows python to call functions in liblutf_agent.so
4. `clutf_global.py` and `_clutf_global.so` : glue code that allows python to call functions in liblutf_global.so
5. `lnetconfig.py and _lnetconfig.so` - glue code to allow python test scripts to utilize the DLC interface.

The build process will check if `python 3.6` and `SWIG 3.0` or higher is installed before building. If these requirements are not met the LUTF will not be built

If the LUTF is built it will be packaged in the `lustre-tests` rpm and installed in `/usr/lib64/lustre/tests/lutf`.

Tasks

Task	Description

C infrastructure	• lutf binary
	• listener thread
	• Heart beat
	• python integration
	  ◦ Look into having a choice between python 3.x and python 2.7.x
	• IPC
	  ◦ Manage connections between the master and the agents
	  ◦ Track the agents
	  ◦ Provide APIs for Request/Response Pair
	  ▪ These APIs will block in the calling thread until a response is received
	  ▪ TODO: What happens if we're calling these APIs from separate Python threads?
	  • What I'm trying to get at is to see how a script can spawn python threads. These threads can do RPC. While the main test thread can continue doing other test logic.
	• API for managing and querying the state kept by the C infrastructure
	  ◦ agent information

SWIG	• SWIG infrastructure to call C APIs
	  ◦ liblnetconfig
	  ◦ LUTF Agent Management
	  ◦ LUTF RPC

lutf.sh	• Spawn the master and agents appropriately • Pass to the master the suite or specific test to run. If nothing is provided all suites are run. • Waits on the master until it exits after running the tests
lutf Python Library	• Association between Agents and node roles (MGS/MDD/etc) ◦ IE build a view of the clustre as identified by the provided environment variables. • API for querying the Agents • Automatically loaded and initialized • API for suites and scripts management and execution • Use the lutf Provisioning Library to clean the clustre before running each test.

lutf Provisioning Library	• API to provision LNet and lnet_selftest
	• API to provision the Lustre File System
	  ◦ API should take a dictionary of the different nodes and based on the node types it spawns a simple File system
	• Both APIs can be used together.
	  ◦ use the LNet provisioning API to provision and configure LNet
	  ◦ use the Lustre FS provisioning API to provision the File system on top of the configured LNet
	• API to un-provision a clustre described in a python dictionary

lutf logging infrastructure	• Set lustre logging levels • Collect lustre logs • collect syslogs • Provide debugging level infrastructure for the test scripts (probably just use the provided Python logging) • API for storing YAML results.

OLD INFORMATION

TODO: Below is old information still being cleaned up

Test Environment Set-Up

Each node which will run the LUTF will need to have the following installed

1. ncurses library
    a. `yum install ncurses-devel`
2. readline library
    a. `yum install readline-devel`
3. python 2.7.5
    a. `https://www.python.org/download/releases/2.7.5/`
    b. `./configure --prefix=<> --enable-shared # it is recommended to install in standard system path`
    c. `make; make install`
4. setuptools
    a. https://pypi.python.org/pypi/setuptools
    b. The way it worked for me:
        i. Download package and untar
        ii. `python2.7 setup.py install`
5. psutils
    a. https://pypi.python.org/pypi?:action=display&name=psutil
        i. untar
        ii. cd to untared directory
        iii. `python2.7 setup.py install`
6. netifaces
    a. https://pypi.python.org/pypi/netifaces
7. Install PyYAML
    a. pip isntall pyyaml

The LUTF will also require that passwordless ssh is setup for all the nodes which run the LUTF. This task is already done when the AT sets up the test cluster.

LUTF Configuration Files

Setup YAML Configuration File

This file is passed to the lutf_perform_test.py. It describes the test system so that the LUTF can be deployed correctly.

```
config:
    type: test-setup
        master: <ip of master>
        agent:
                0: <ip of 1st agent>
                1: <ip of 2nd agent>
                ...
                N: <ip of Nth agent>
    master_cfg: <path to master config file>
        agent_cfg: <path to agent config file>
        test_cfg: <path to test config file>
        result_dir: <path to the directory to store the test results in>
```

Master YAML Configuration File

This configuration file describes the information the master needs in order to start

```
config:
   type: master
   mport: <OPTIONAL: master port. Default: 8494>
   dport: <master daemon port. Used to communicate with master>
   base_path: <OPTIONAL: base path to the LUTF directory.
               Default: /usr/lib64/lustre/tests>
   extra_py: <OPTIONAL: extra python paths>
```

Agent YAML Configuration File

This configuration file describes the information the agent needs in order to start

```
config:
   type: agent
   maddress: <master address>
   mport: <OPTIONAL: master port. Default: 8494>
   daemon: <1 - run in daemon mode. Defaults to 0>
   base_path: <OPTIONAL: base path to the LUTF directory
                          Default: /usr/lib64/lustre/tests>
   extra_py: <extra python paths>
```

The agent's maddress can be inserted automatically, since it's already defined in the setup configuration file.

Both the Master and Agent configuration files can be optional. If nothing is provided all the parameters will be defaulted. In the absence of an agent configuration file one will be automatically created that only has the maddress field. Example below:

```
config:
   type: agent
   maddress: <master address as provided in the setup file>
```

Test YAML Configuration File

This configuration file describes the list of tests to run

```
config:
   type: tests-scripts
   testsID: <test id>
   timeout: <how long to wait before the test is considered a failure.
             If not provided then the script will wait until killed by
                         the AT. If a list of tests are provided the time the
                         script will wait will be timeout * num_of_tests>
   tests:
        0: <test set name>
        1: <test set name>
        2: <test set name>
        ....
        N: <test set name>

# "test set name" is the name of the directory under lutf/python/tests
# which includes the tests to be run. For example: dlc, multi-rail, etc
```

LUTF Result file

This YAML result file describes the results of the tests that were requested to run (**TODO**: it's not clear exactly what the result file will look like. What definitely will be needed is the results zip file generated by the LUTF master. This will need to be available from Maloo to be able to understand which tests failed, and why)

```
TestGroup:
    test_group: review-ldiskfs
    testhost: trevis-13vm5
    submission: Mon May  8 15:54:41 UTC 2017
    user_name: root
autotest_result_group_id: 5e11dc5b-7dd7-48a1-b4a3-74a333acd912
test_sequence: 1
test_index: 10
session_group_id: cfeff6b3-60fc-438a-88ef-68e65a08694f
enforcing: true
triggering_build_number: 45090
triggering_job_name: lustre-reviews
total_enforcing_sessions: 5
code_review:
 type: Gerrit
 url: review.whamcloud.com
 project: fs/lustre-release
 branch: multi-rail
 identifiers:
 - id: 3fbd25eb0fe90e4f34e36bad006c73d756ef8499
issue_tracker:
 type: Jira
 url: jira.hpdd.intel.com
 identifiers:
 - id: LU-9119
Tests:
- name: dlc
        description: lutf dlc
        submission: Mon May  8 15:54:43 UTC 2017
        report_version: 2
        result_path: lustre-release/lustre/tests/lutf/python/tests/
        SubTests:
        - name: test_01
          status: PASS
          duration: 2
          return_code: 0
          error:
        - name: test_02
          status: PASS
          duration: 2
          return_code: 0
          error:
        duration: 5
        status: PASS
-  name: multi-rail
        description: lutf multi-rail
        submission: Mon May  8 15:59:43 UTC 2017
        report_version: 2
        result_path: lustre-release/lustre/tests/lutf/python/tests/
        SubTests:
        - name: test_01
            status: PASS
            duration: 2
            return_code: 0
            error:
        - name: test_02
            status: PASS
            duration: 2
            return_code: 0
            error:
        duration: 5
        status: PASS
```

LUTF Master API

There are two ways to start the LUTF Master.

1. In interactive mode

    a. This is useful for interactive testing
2. In Daemon mode
    a. This is useful for automatic testing

In either of these modes the Master instance can process the following requests:

1. Query the status of the LUTF master and its agents
2. Run tests
3. Collect results

A C API SWIG wrapped to allow it to be called from python will be provided. The API will send messages to the identified LUTF Master instance to perform the above tasks, and then wait indefinitely until the request completes.

Query Status

1. Send a QUERY message to the LUTF Master
2. LUTF Master will look up all the agents currently connected.
3. LUTF Master will bundle the information and send it back.
4. The result is examined against expected values
5. The script succeeds or fails.

Run Tests

1. Send a RUN_TESTS message to the LUTF Master
2. Include a buffer containing the YAML block identifying the tests to run
3. LUTF master will run the tests
    a. For each individual test run a result file is generated
    b. An overall test run result file will also be generated
4. Once the LUTF master finishes running the tests it will ZIP up the results and return a path to the results to the caller.
5. The script will then collect the results

Collect Results

1. Send a COLLECT_RESULTS with the test ID to collect
2. LUTF Master ZIP up the test results and returns back to caller.
3. The script can then collect the results.

Message Structure

```
typedef enum {
        EN_MSG_TYPE_HB = 0,
        EN_MSG_TYPE_GET_NUM_AGENTS,
        EN_MSG_TYPE_MAX
} lutf_msg_type_t;


typedef struct lutf_message_hdr_s {
        lutf_msg_type_t type;
        unsigned int len;
        struct in_addr ip;
        unsigned int version;
} lutf_message_hdr_t;
```

YAML Response

For each of the three requests identified above, the LUTF Master will respond with a YAML block. The python script can use the python YAML parser to extract relevant information.

```
master_response:
        status: <[Success | Failure]>
    agents:
                - name: <agent name>
                  ip: <agent ip address>
                - name: <agent name>
                  ip: <agent ip address>
    test_results: <path to zipped results>
```

Network Interface Discovery

The LUTF test scripts will need to be implemented in a generic way. Which means that each test scripts which requires the use of interfaces, will need to discover the interfaces available to it on the node. If there are sufficient number of interfaces of the correct type, then the test can continue otherwise the test will be skipped and reported as such in the final result.

Maloo

1. A separate section is to be created in Maloo to display LUTF test results.
2. The results from output YAML file passed from AT are displayed in the LUTF results section.
3. A Test-parameter specifically for LUTF tests to be defined that will allow to run only LUTF tests. LUTF tests will only run if that Test-parameter is set. There is no need to run LUTF for every patch committed.

Improvements

1. Currently the LUTF is designed to have the Python infrastructure establish a Telnet connection to facilitate Master to scp the test scripts to Agent and then execute those test scripts. The Telnet approach can be improved upon by using SSH instead.
2. A **synchronization mechanism** can be added to synchronize the different parts of one test script running on different Agents by providing an API that uses notification mechanism. The Master node will control this synchronization between different Agent nodes that are used for running a test script. An example scenario of how it would be implemented is -If a test script is such that it requires to do some operation on more than one Agent node, then as one part of a test script runs to it completion on one Agent, it would notify the Master about its status by calling this API and then Master can redirect this event to the main script waiting on it which will trigger the other part (operation) to start execution on another Agent node.

Misc

Some Sample files from Auster

A sample Config file used by Auster	A sample result YAML file from Auster
sample.sh	results.yml

Another proposal to passing information to the LUTF if it can not be passed via a YAML config file as described above.

```
#!/bin/bash
#Key Exports
export master_HOST=onyx-15vm1
export agent1_HOST=onyx-16vm1
export agent2_HOST=onyx-17vm1
export agent3_HOST=onyx-18vm1
export AGENTCOUNT=3

VERBOSE=true

# ports for LUTF Telnet connection
export MASTER_PORT=8494
export AGENT_PORT=8094

# script and result paths
script_DIR=$LUSTRE/tests/lutf/python/test/dlc/
output_DIR=$LUSTRE/tests/lutf/python/tests/
```