LNet Health Test Plan

System Setup

LNet Resiliency/Health is mainly implemented at the LNet layer. The LND layer is only responsible for propagating specific errors up to the LNet Layer, which then reacts to those errors as defined in the Requirements and HLD documentation.

In order to properly test this feature fine grained control over the LND behavior is required. The drop/delay message policies will be modified to simulate the various errors that could occur when sending a message. This is described in the sections below.

Selection Algorithm Scenarios

Test #	Tag	Procedure	Script	Result
1	SRC_SPEC_LOCAL_MR_DST	• MR Node • MR Peer • Send a ping • REPLY for PING should always come on the same interface that PING was sent on. • Check the TRACE in the logs to verify • Repeat the test. A different local NI should be used for each new PING.		pass
2	SRC_SPEC_LOCAL_MR_DST	• MR Node • MR Peer • Initiate discovery • Node PING Peer • Node PUSH Peer • Node should respond with an ACK to the same interface as the one it received the PUSH on • Check the TRACE in the logs to verify • Repeat the test ◦ Peer's local_ni when sending the PUSH should be different.		pass
3	SRC_SPEC_ROUTER_MR_DST	• MR Node • NMR Router • MR Peer • Send a ping • REPLY for PING should always come on the same interface that PING was sent on. • Check the TRACE in the logs to verify • Router should be used • Repeat the test. A different local NI should be used for each new PING.		pass
4	SRC_SPEC_ROUTER_MR_DST	• MR Node • NMR Router • MR Peer • Initiate discovery • Node PING Peer • Node PUSH Peer • Node should respond with an ACK to the same interface as the one it received the PUSH on • Check the TRACE in the logs to verify • Router should be used • Repeat the test. Peer's local_ni when sending the PUSH should be different.		pass
5	SRC_SPEC_ROUTER_MR_DST	• MR Node • MR Router • MR Peer • Send a ping • REPLY for PING should always come on the same interface that PING was sent on. • Check the TRACE in the logs to verify • Repeat sending • Router interfaces should be used in round robin, while the peer destination should remain constant. • Repeat the test. A different local NI should be used for each new PING.		pass

6	SRC_SPEC_ROUTER_MR_DST	• MR Node • MR Router • MR Peer • Initiate discovery • Node PING Peer • Node PUSH Peer • Node should respond with an ACK to the same interface as the one it received the PUSH on • Check the TRACE in the logs to verify • Router interfaces should be used in round robin, while the peer destination should remain constant. • Repeat the test. Peer's local_ni when sending the PUSH should be different.		pass
7	SRC_SPEC_LOCAL_NMR_DST	• Same as 1 and 2 • Except that repeating the test will not result in a different local_ni being used.		pass
8	SRC_SPEC_ROUTER_NMR_DST	• Same as 3 - 6 • Except that repeating the test will not result in different local NIs being used.		pass
9	SRC_ANY_LOCAL_MR_DST	• MR Node • MR Peer • Send multiple PINGs • PING REPLYs should come on the same interface • Every PING will select a new local/remote NIs		pass
10	SRC_ANY_ROUTER_MR_DST	• MR Node • NMR Router • MR Peer • Send Multiple PINGs • Node will cycle over local_NIs • Node will use the same destination NID as final destination • Node will use the NMR Router		pass
11	SRC_ANY_ROUTER_MR_DST	• MR Node • MR Router • MR Peer • Send Multiple PINGs • Node will cycle over local_NIs • Node will use the same destination NID as final destination • Node will use the different interfaces of the MR Router • MR Router will cycle over the interfaces of the Final destination.		pass
12	SRC_ANY_LOCAL_NMR_DST	• MR Node • NMR Peer • Send multiple PINGs • Node will use same source/dst NID for all PINGs		pass
13	SRC_ANY_ROUTER_NMR_DST	• MR Node • NMR Router • NMR Peer • Send multiple PINGs • Node will use the same source/dst NIDs for all PINGs • Node will use the router interface		pass
14	SRC_ANY_ROUTER_NMR_DST	• MR Node • MR Router • NMR Peer • Send multiple PINGs • Node will use the same source/dst NIDs for all PINGs • Node will cycle through the Router's interfaces		pass

Error Scenarios

Synchronous Errors

Test #	Tag	Procedure	Script	Result
1	Immediate Failure	• Send a PING • simulate an immediate LND failure (EX: NOMEM) • Message should not be resent	lnetctl discover <nid> lctl net_drop_add with "-e local_error" lnetctl discover <nid>	pass

Asynchronous Errors

Test #	Tag	Procedure	Script	Result
1	LNET_MSG_STATUS_LOCAL_INTERRUPT LNET_MSG_STATUS_LOCAL_DROPPED LNET_MSG_STATUS_LOCAL_ABORTED LNET_MSG_STATUS_LOCAL_NO_ROUTE LNET_MSG_STATUS_LOCAL_TIMEOUT	• MR Node with Multiple interfaces • Send a PING • Simulate an <error> • PING msg should be queued on resend queue • PING msg will be resent on a different interface • Failed interfaces' health value will be decremented • Failed interface will be placed on the recovery queue	Examples: lctl net_drop_add -s 10.9.10.3@tcp -d 10.9.10.4@tcp -m GET -i 20 -e local_dropped Key messages in debug log: (lib-msg.c:762:lnet_health_check()) 10.9.10.3@tcp->10.9.10.4@tcp:GET: LOCAL_DROPPED - queuing for resend (lib-msg.c:508:lnet_handle_local_failure()) ni 10.9.10.3@tcp added to recovery queue. Health = 950 (lib-move.c:2928:lnet_recover_local_nis()) attempting to recover local ni: 10.9.10.3@tcp	pass
2	Sensitivity == 0	• Same setup as 1 • NI is not placed on the recovery queue		pass
3	Sensitivity > 0	• Same setup as 1 • NI is placed on the recovery queue • Monitor network activity as NI is pinged until health is back to maximum		pass
4	Sensitivity > 0 Buggy interface	• Same setup as 1 • NI is placed on recovery queue • NI is pinged ever 1 second • Simulate ping failure ever other ping • NI's health should be decremented on failure • NI should remain on the recovery queue		
5	Retry count == 0	• Same setup as 1 • Message will not be retried and the message will be finalized immediately		pass
6	Retry count > 0	• Same setup as 1 • Message will be transmitted for a maximum of retry count or until the message expires	Key messages in debug log: (lib-move.c:1715:lnet_handle_send()) TRACE: 10.9.10.3@tcp(10.9.10.3@tcp:<?>) -> 10.9.10.4@tcp(10.9.10.4@tcp:10.9.10.4@tcp) : GET try# 0 (lib-move.c:1715:lnet_handle_send()) TRACE: 10.9.10.3@tcp(10.9.10.3@tcp:<?>) -> 10.9.10.4@tcp(10.9.10.4@tcp:10.9.10.4@tcp) : GET try# 1	pass
7	REPLY timeout	• Same setup as 1 • Except Use LNet selftest • Simulate a local timeout • Re-transmit • No REPLY received • Message is finalized and TIMEOUT event is propagated.		pass
8	ACK timeout	• Same setup as 7 except simulate ACK timeout		pass

9	LNET_MSG_STATUS_LOCAL_ERROR	• Same setup as 1 • Message is finalized immediately (not resent) • Local NI is placed on the recovery queue • Same procedure to recover the local NI		pass
10	LNET_MSG_STATUS_REMOTE_DROPPED	• Same setup as 1 • Message is queued for resend depending on retry_count • peer_ni is placed on the recovery queue (not if sensitivity == 0) • peer_ni is pinged every 1 second		pass
11	LNET_MSG_STATUS_REMOTE_ERROR LNET_MSG_STATUS_REMOTE_TIMEOUT LNET_MSG_STATUS_NETWORK_TIMEOUT	• Same setup as 1 • Message is not resent • peer_ni recovery happens as outlined in previous cases		pass

Random Failures

Test #	Tag	Procedure	Script	Result
1	self test	• MR Node • NMR Peer • Self-test • Randomize local NI failure • Randomize Remote NI failure	ip link set eth1 down	pass
2	self test	• MR Node • MR Peer • Self-test • Randomize local NI failure • Randomize Remote NI failure		pass
3	self test	• MR Node • MR Router • NMR Peer • Self-test • Randomize local NI failure • Randomize Remote NI failure		
4	self test	• MR Node • MR Router • MR Peer • Self-test • Randomize local NI failure • Randomize Remote NI failure		
5	self test	• MR Node • NMR Router • NMR Peer • Self-test • Randomize local NI failure • Randomize Remote NI failure		
6	self test	• MR Node • NMR Router • MR Peer • Self-test • Randomize local NI failure • Randomize Remote NI failure		

MR Router Testing

Test #	Tag	Procedure	Script	Result

1	Discovery triggered on route add	• Bring up Router A with two interfaces
		  ◦ tcp0
		  ◦ tcp1
		• Bring up Peer A and add network on tcp0
		• Add router to tcp1 on peerA
		• Observe that a discovery occurs from peer A Router A
				pass

2	Discovery triggered on interval	• Bring up Router A with two interfaces
		  ◦ tcp0
		  ◦ tcp1
		• Bring up Peer A and add network on tcp0
		• Add router to tcp1 on peerA
		• Observe that a discovery occurs from peer A Router A
		• Keep the two nodes up for 4 minutes
		• Every router_interval_timeout a discovery should occur from peerA RouterA
				pass

3	Router tcp1 down due to no traffic	• Bring up Router A with two interfaces
		  ◦ tcp0
		  ◦ tcp1
		• Bring up Peer A and add network on tcp0
		• Add router to tcp1 on peerA
		• Observe that a discovery occurs from peer A Router A
		• Keep the two nodes up for 4 minutes
		• Every router_interval_timeout a discovery should occur from peerA RouterA
		• Since there is no traffic on tcp1 RouterA tcp1 should be down
		  ◦ verify via: lnetctl net show -v
				pass

4	Router tcp1 comes up when peerB is brought up	• Bring up Router A with two interfaces
		  ◦ tcp0
		  ◦ tcp1
		• Bring up Peer A and add network on tcp0
		• Add router to tcp1 on peerA
		• Observe that a discovery occurs from peer A Router A
		• Keep the two nodes up for 4 minutes
		• Every router_interval_timeout a discovery should occur from peerA RouterA
		• Since there is no traffic on tcp1 RouterA tcp1 should be down
		  ◦ verify via: lnetctl net show -v
		• Bring up Peer B and add network on tcp1
		• Add router to tcp on peer B
		• Observe that a discovery occurs from peerB RouterA
		• Observe that a RouterA tcp1 is now up
				pass

5	Add route without router there	• Bring up Peer A and add network on tcp0 • Add route to tcp1 on peerA • Observe that a discovery occurs but no response since router is not up • lnetctl route show -v # shows that router is down • lnetctl peer show -v # shows the peer is down • Bring up Router A with two interfaces: tcp0, tcp1 • After router_interval_timeout a discovery should verify that router A is up • lnetctl route show -v # shows that router is down because no routerA tcp1 network should be down • lnetctl peer show -v # shows the peer is up • Bring up PeerB and add network on tcp1 • lnetctl route show -v # shows that router is up		pass

6	traffic should trigger an attempt at router discovery	• Bring up Peer A and add network on tcp0
		• Add route to tcp1 on peerA
		• Observe that a discovery occurs but no response since router is not up
		• lnetctl route show -v # shows that router is down
		• lnetctl peer show -v # shows the router is down
		• Bring up Router A with two interfaces: tcp0, tcp1
		• Bring up PeerB and add network on tcp1
		• Before the router_interval_timeout expires do a:
		  ◦ lnetctl discover Router@tcp
		  ◦ This should trigger a discovery of router A
		  ◦ lnetctl peer show -v # shows the peer is up and multi-rail
		  ◦ lnetctl route show -v # shows the route up
				pass

7	Ping should not trigger discovery of router	• Bring up Peer A and add network on tcp0
		• Add router to tcp1 on peerA
		• Observe that a discovery occurs but no response since router is not up
		• lnetctl route show -v # shows that router is down
		• lnetctl peer show -v # shows the router is down
		• Bring up Router A with two interfaces: tcp0, tcp1
		• Bring up PeerB and add network on tcp1
		• Before the router_interval_timeout expires do a:
		  ◦ lnetctl ping PeerB@tcp1
		  ◦ This should NOT trigger a discovery of router A
		  ◦ ping should fail
		  ◦ lnetctl peer show -v # shows the peer is down
		  ◦ lnetctl route show -v # shows the route down
				pass

8	Multi-interface router even traffic distribution	• Bring up Router A with 4 interfaces. 2 on tcp0 and 2 on tcp1 • Bring up Peer A with interface on tcp0 • Bring up Peer B with interface on tcp1 • Run traffic using selftest • Observe that traffic is distributed on all router interfaces evenly		pass

9	Multi-interface router with one bad interface	• Bring up Router A with 4 interfaces. 2 on tcp0 and 2 on tcp1 • Bring up Peer A with interface on tcp0 • Bring up Peer B with interface on tcp1 • Run traffic using selftest • Observe that traffic is distributed on all router interfaces evenly • Enable health (sensitivity, retries) • Add a PUT drop rule on the router to drop traffic on one of the interfaces in tcp0 • Observe that traffic goes to the other interfaces. There shouldn't be any drop in traffic. • As long as the interface has less than optimal health, it should never be used for routing.		pass
10	Multi-interface router with a bad interface that recovers	• Bring up Router A with 4 interfaces. 2 on tcp0 and 2 on tcp1 • Bring up Peer A with interface on tcp0 • Bring up Peer B with interface on tcp1 • Run traffic using selftest • Observe that traffic is distributed on all router interfaces evenly • Enable health (sensitivity, retries) • Add a PUT drop rule on the router to drop traffic on one of the interfaces in tcp0 • Observe that traffic goes to the other interfaces. There shouldn't be any drop in traffic. • As long as the interface has less than optimal health, it should never be used for routing. • Remove the PUT drop rule from the router • Eventually that interface should be healthy again • Traffic should resume using that interface		pass In an idle system the bad peer interface will be pinged once every second causing its sequence number to go up. So when it comes back online it will not be used until the sequence numbers equalize. This will be the case if the system is busy, but the issue will be reversed.
11	Multi-Router /Multi-interface setup	• Bring up router A with 4 interfaces 2 on tcp0 and 2 on tcp1 • Bring up router B with 4 interfaces 2 on tcp0 and 2 on tcp1 • Bring up Peer A with interface on tcp0 • Bring up Peer B with interface on tcp1 • Run traffic • Observe that traffic is distributed evenly on the interfaces of router A and B		pass
12	Multi-Router /Multi-interface setup with failed gateway	• Bring up router A with 4 interfaces 2 on tcp0 and 2 on tcp1 • Bring up router B with 4 interfaces 2 on tcp0 and 2 on tcp1 • Bring up Peer A with interface on tcp0 • Bring up Peer B with interface on tcp1 • Run traffic • Observe that traffic is distributed evenly on the interfaces of router A and B • Shutdown router A • Observe that traffic is diverted to Router B with no drop in traffic.		pass

13	Multi-Router /Multi-interface setup with router recovery	• Bring up router A with 4 interfaces 2 on tcp0 and 2 on tcp1 • Bring up router B with 4 interfaces 2 on tcp0 and 2 on tcp1 • Bring up Peer A with interface on tcp0 • Bring up Peer B with interface on tcp1 • Run traffic • Observe that traffic is distributed evenly on the interfaces of router A and B • Shutdown router A • Observe that traffic is diverted to Router B with no failure. • Startup Router A • Observe that traffic starts going through Router A again. There should be no drop in traffic		Problem found. Possibly with discovery. 1. bring up two routers with 4 interfaces 2 on each network 2. bring down one of the routers 3. bring it up again but with only 2 of its interfaces on 1 network 4. Client goes berserk, keeps trying to discover it. toggles between state: 0x139 and 39 There were a couple of issues here: 1. the sequence numbers were getting misalligned when the router was brought down. This caused discovery not to work correctly 2. We restricted the router peer-NIs from being deleted. But we need to differentiate between configuration changes and discovery changes. The eariler should not allow deleting peer_nis from routers unless the route is removed first. The latter should allow peer updates because the peer itself is giving us new information. Pass
14	router sensitivity < 100	• Bring up router A with 4 interfaces 2 on tcp0 and 2 on tcp1 • Bring up Peer A with interface on tcp0 • Bring up Peer B with interface on tcp1 • modify the router_sensitivity to 50% • Add a drop rule on router A • Observe that traffic doesn't completely stop to Router A until its health goes to 50% of the optimal value.		
15	Extra Health Testing	• Run through the health test cases above while there exists a multi-rail router.		

User Interface

Test #	Tag	Procedure	Script	Result

1	lnet_transaction_timeout	• Set lnet_transaction_timeout to a value < retry_count via lnetctl and YAML
		  ◦ This should lead to a failure to set
		• Set lnet_transaction_timeout to a value > retr_count via lnetctl and YAML
		  ◦ lnet_lnd_timeout value should == lnet_transaction_timeout / retry_count
		• Show value via "lnetctl global show"
			lnetctl set transaction_timeout <value>	pass

2	lnet_retry_count	• Set the lnet_retry_count to a value > lnet_transaction_timeout via lnetctl and YAML
		  ◦ This should lead to a failure to set
		• Set the lnet_retry_count to a value < lnet_transaction_timeout via lnetctl and YAML
		  ◦ lnet_lnd_timeout value should == lnet_transaction_timeout / retry_count
		• Show value via "lnetctl global show"
			lnetctl set retry_count <value>	pass

3	lnet_health_sensitivity	• Set the lnet_health sensitivity from lnetctl and from YAML • Show value via "lnetctl global show"	lnetctl set health_sensitivity <value>	~~LU-11530~~ - lnetctl set health_sensitivity does not return error with value greater than 1000 RESOLVED
4	NI statistics	• verify LNet health statistics ◦ lnetctl net show -v 3		pass
5	Peer NI statistics	• verify LNet health statistics for peer NIs ◦ lnetctl peer show -v 3		pass

6	NI Health value	• verify setting the local NI health statistics ◦ lnetctl net set --nid <nid> --health <value> • Redo from YAML		LU-11529 - lnetctl does not change NI health value from YAML configuration file OPEN
7	Peer NI Health value	• verify setting the local NI health statistics ◦ lnetctl peer set --nid <nid> --health <value> • Redo from YAML		LU-11529 - lnetctl does not change NI health value from YAML configuration file OPEN

Testing Tools

The drop policy has been modified to drop outgoing messages with specific errors. This can be done via the following commands. Unfortunately, for details on these commands you'll need to look at the code. A combination of these commands on the different nodes should cover approximately 75% of the health code paths.

```
lctl net_drop_add -s *@tcp -d *@tcp -m ACK -i 20
lctl net_drop_add -s *@tcp -d *@tcp -m REPLY -i 20
lctl net_drop_add -s *@tcp -d *@tcp -m GET -i 43 -e random -n
lctl net_drop_add -s *@tcp -d *@tcp -m PUT -i 20 -e random
lctl net_drop_del -s *@tcp -d *@tcp
```

The `-e` parameter can take the following arguments

```
local_interrupt                # will result in a resend
local_dropped                   # will result in a resend
local_aborted                   # will result in a resend
local_no_route                   # will result in a resend
local_error                           # will not result in a resend
local_timeout                   # will result in a resend
remote_error                   # will not result in a resend
remote_dropped                   # will result in a resend
remote_timeout                   # will not result in a resend
network_timeout                   # will not result in a resend
random
silent_queue                   # will queue the message and never call lnet_finalize()
```

The `-e` can be repeated multiple times to specify a set of different errors to select randomly from. `random` can be given to `-e` to select any error to simulate at random.

Simulation Details

The error simulation occurs immediately before putting the message on the wire. If the drop rule policy is defined and is matched, then the message is not sent and error is simulated.

Types of Error Simulation Testing

Type	Description
Drop with error	This is the newly added error simulation. And it is designed to simulate different health failures. This can be used to exercise the following scenarios 1. re-transmit message scenair 2. Immediate failure and message finalization 3. Local NI recovery 4. Peer NI recovery 5. Finalizing active message
Drop Received messages	This an existing rule and it can be used to drop received GET/PUT messages. This will result in no ACK/REPLY being sent to the message initiator and will exercise the response timeout code.

Queuing messages	In a heavily used systems, especially routers, the credits can dip below zero and a message can be queued internally. It's possible that these message can be queued for a long period of time, so a mechanims was created to finalize these messages after the expiration time. A command can be issued to simulate queueing. This will queue the message on a separate queue which is never checked except when the message expires. The message credit is not returned until the message expires.

Configuration

LNet Health is off by default. To turn it on, two configuration parameters need to be set

1. Retry counter. This will indicate how many times a message should be resent before it succeeds or times out. It default to 0, which means no message re-transmission will occur.
    a. `lnetctl set retry_count <value>`
2. Health sensitivity. This is a value by which to decrement the health of the interface. When an interface health value goes below the optimal value, it gets placed on a recovery queue and will be pinged every second until its health recovers. This value by default is 0, which means that a peer or local NI will never go into recovery state.
    a. `lnetctl set health_sensitivity <value>`
3. Transaction timeout. This is the timeout value to wait before a response expires or before a message on the active list expires.
    a. `lnetctl set transaction_timeout <value>`