Adaptive RPC Timeouts

Eric Barton, Nathan Rutman

March 28, 2007

1 Introduction

Lustre clients use RPC timeouts to trigger recovery strategies. A fixed timeout is sufficient to detect server failure promptly if RPC round-trip times are reasonably constant. However this is not the case in clusters with many thousands of nodes since the average time a server takes to handle an RPC scales approximately linearly with the number of clients contending for it. In this situation a fixed timeout has to be set high enough to account for worst-case congestion, making prompt server failover impossible, even under light load.

2 Functional Specification

1. Scale the client's RPC timeout with server congestion.

The main aim of adaptive timeouts is to distinguish server congestion from server death. As servers become congested, RPC round-trip times increase, and the client's timeout must grow to match. Similarly, as server load decreases, RPC round-trip times decrease and the client's timeout must reduce.

2. Scale the server's lock callback timeout with network congestion and client progress.

With network congestion or slow server response, a lock called back from a client may require more time than the server callback timeout allows. These should also use adaptive timeouts.

3. No significant negative impact on performance under all cluster loads.

Adaptive timeouts must not deliver prompt failover at the expense of sustained performance or substantially increased cluster boot times. For example, additional message passing used to implement adaptive timeouts must not exacerbate network congestion. While it may be acceptable for individual client startup times to increase measurably, these delays must not serialize between clients.

4. Integrate with lustre recovery strategy.

The RPC timeout is not the only component of lustre recovery - it simply signals the likelihood that further RPCs to the same server may fail. Network failure on one path to the server could be the real cause of the timeout and lustre recovery should therefore not immediately assume that a timeout means that the server is unreachable.

5. No impact on LNET in the first implementation.

Adaptive timeouts should account for additional latencies introduced by network loading and router congestion, and can be accounted for at the ptlrpc level. Changes to LNET (e.g. true out-of-band signaling or complete integration of LNET and lustre timeouts) will be avoided to ensure a timely implementation.

6. Minimize site tuning.

Adaptive timeouts should reduce the need for site tuning either by eliminating tuneables or setting defaults which are appropriate across the widest possible cluster types and configurations.

3 Use Cases

3.1 Heavy load

A large cluster under heavy loading causes servers to get "backed up" processing RPC requests. The server realizes that it is unlike to be able to reply within the client's stated timeout and sends an early reply to the client giving a more accurate estimatation of completion time. The client will then wait that amount of time for a response, rather than an arbitrary fixed value.

3.2 Slow lock callback

A lock revocation callback on a client requires the client to flush some large buffers. The I/O is proceeding slowly (due to e.g. heavy server loading). The client sends an early reply back to the server stating its estimated completion time.

3.3 Recovery

Recovery times for large clusters have been slow due to large obd_timeout values, which were required to handle server congestion. Adaptive timeouts decouple the network and server loading from the obd_timeout value, leading to its reduction back down to a more responsive value and subsequently faster recovery.

4 Logic Specification

The initial implementation adapts RPC timeouts and lock callback timeouts to become independent of server and network performance. Timeout tuning may still be required to account for network congestion at the server and on LNET routers. However this should be the exception rather than the rule - experience shows that network latency is much lower than current RPC round-trip times on a congested cluster.

There are three basic mechanisms. Firstly, the client bases its RPC timeout on recent RPC round-trip times and also on recent RPC service times which the server includes in all replies. Secondly, the server may send a keepalive reply to adjust the client's timeout if it will not be able to deliver the normal RPC reply before the client will time out. Thirdly, if the client times out an RPC, it should fall back to a lighter-weight reconnection strategy rather than assuming the server is actually down. This third mechanism is beyond the scope of this document. The first two are implemented as follows.

(Take "client" to mean the initiator of an RPC request, and "server" to be the handler of that request. E.g. an MDT may be a "client" of an ldlm "server" running on a Lustre compute node.)

- The server maintains a current worst-case RPC service time. RPC service time is measured from when LNET delivers the RPC request (arrival event) to when the server posts the normal RPC reply. The current worst-case RPC service time is the maximum service time computed over the last last N RPCs.

- The initial connection request RPC uses a site-tunable timeout that accounts for worst-case network congestion. This ensures the client does not give up before the server has any hope of replying. The default is good for the vast majority of clusters (e.g. 10 seconds caters for a server with a 1GigE NIC with over 1,000 clients or an XT3 server with nearly 10,000 clients).

- The client keeps 2 logically separate reply buffers - the normal RPC reply and the early "busy" reply. These will be allocated contiguously and posted with a single ME/MD for the lowest performance impact. This requires a change in the ptlrpc layer to allow it to unlink the ME/MD only when the normal reply is received.

- The client computes an RPC service timeout based on the most recent RPC service times it has seen. This could use a tunable factor (e.g. 110%) and/or a tunable increment (e.g. 2 seconds) - the defaults should be good for all sites. This timeout is sent in the RPC request. The RPC timeout is computed by adding the service timeout to the worst-case network congestion timeout. The client must maintain a history of RPC service times for each portal on each server.

- The server checks the service timeout in the RPC request when it arrives. If it is less than the server's current maximum, an early reply message is sent immediately stating the server's current value.

- Both client and server's RPC service timeouts are visible via /proc variables. The client also publishes its current and worst-ever network latencies (i.e. round-trip time minus RPC service time).

5 State Management

There is no persistant data; obd_timeout will be used for initial values before any measured data is known.

Servers maintain a worst-case RPC service time for each service.

Clients maintain an estimated max service time per portal per import

ME/MD must now accomodate two reply messages, so we must maintain a pointer to the real reply.

5.1 obd_timeout

Decoupling the RPC timeouts from obd_timeout will lead to a much less deterministic sequence of timeouts. Notably, if the adaptive timeouts grow larger than the fixed obd_timeout, it is quite possible that processes will give up too soon, even though the RPCs will successfully complete. We make the assumption in code that if we have waited longer than obd_timeout, that something failed or will fail – this is no longer the case. We also make the assumption that some things (ldlm_callback) should take much less time than a "full" obd_timeout - this is also no longer the case with adaptive timeouts, as the regular RPC timeout may now be very short, on the same order as an ldlm timeout.

Therefore, we will have to replace many of the obd_timeout based wait intervals with an appropriate adaptive interval (based on the import or export measured latencies). All multipliers should be removed, since they are meaningless with decoupled timings.

- ldlm_timeout = ot/3, ADAPTIVE
- PING_INTERVAL - ot/4. The relevancy of the pinger becomes less clear with adaptive timeouts (the ping_evictor may still be useful). The interval should remain constant.
- RECONNECT_INTERVAL - ot/10, remove throttling in import_select_conn.
- OBD_RECOVERY_TIMEOUT - ot*5/2. Old clients may be slow to reconnect (are using a large adaptive timeout). For this case, track the maximum timeout reported in the reconnect attempts and reset the recovery completion timer based on this value every time a client rejoins.
- ptlrpc_connect_import (initial connect) = ot/20, ADAPTIVE
- ldlm_completion_ast - ot, ADAPTIVE
- __ldlm_add_waiting_lock - ot/2, ADAPTIVE
- fsfilt_check_slow - ot/2, DISK_TIMEOUT
- filter_precreate - ot/4, DISK_TIMEOUT
- mds_sendpage - ot/4, ADAPTIVE

- ptlrpc_invalidate_import - ot, use the timeout at the head of the imp_sending_list; keep resetting the timeout as long as the head keeps changing.

6 Alternatives

Calculation of the server times and the client times may have to be adjusted - e.g.:

- use moving-average + n*stdev instead of worst-case
- clients timeout is some function of the reported server time and the measured total RPC round-trip time

7 Focus for inspection

- insure timeouts are robust against 0/infinite/failed cases.
- insure unit test gives good coverage.
- insure a thread is available for sending early replies under all conditions.