NRS Conceptual Design

Version	Date	Comment	Author
1.0	01/28/2013	Initial revision	Nikitas Angelinas <nikitas_angelinas@xyratex.com>

Table of Contents

1. Introduction

This document presents an overview of the design for the Network Request Scheduler (NRS) subcomponent of PTLRPC for Lustre. The project has been carried out as a natural continuation of a prototype implementation produced by Liang Zhen of Whamcloud, and so no HLD had been authored before development on the project commenced[1]. This document aims to fill the void between previous architectural-level descriptions of NRS and the NRS patches that are currently under review at Intel's Gerrit. Main concepts, APIs and data structures are presented, although these may be subject to some changes as the code reviews progress.

1 To be accurate, an HLD for NRS had been authored by Xyratex, but it describes a different NRS design; this document can be found in the NRS Jira ticket, or relevant discussion thread in the lustre-devel mailing list.

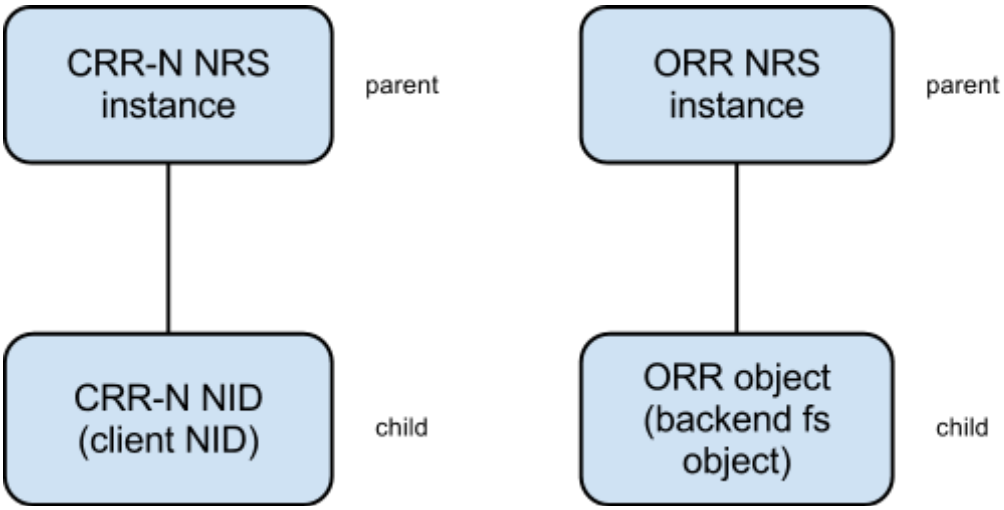
This document has been deliberately kept short and informal in its presentation; interested readers may also find the relative Whamcloud Jira issue, the “NRS HLD” thread in lustre-devel, and previous presentation material of use for further investigation.

2. NRS framework and policies

2.1 Concepts

An **NRS policy** (ptlrpc_nrs_policy) is an implementation of an NRS scheduling algorithm. NRS algorithms are different strategies for managing the flow of RPCs through server nodes, with the aim of achieving various effects. At the time of writing this document, the curerntly authored NRS policies are:
[a] First In, First Out (**FIFO)**; this policy is compatible with all PTLRPC services. FIFO is a logical wrapper around previous, non-NRS PTLRPC functionality, i.e. requests are handled by this policy in the same order as the arrive from LNET. This acts as the default policy for handling all types of RPCs, and also as the fallback policy in case a primary policy (please see below) fails to handle, or denies to handle a request for some reason.
[b] Client Round Robin over NIDs (**CRR-N**); this policy is compatible with all PTLRPC services. CRR-N batch-schedules incoming requests in a round robin manner across client NIDs. This policy aims to provide better load balancing and thus better overall performance across the cluster by trying to respond to client requests fairly. It is likely that it might help shorten job completion times or enable better request merging at lower OS layers.
[c] Object Round Robin (**ORR**); this policy is only compatible with “ost_io” PTLRPC services on OSS nodes. ORR batch-schedules incoming OST_READ and/or OST_WRITE requests in a round robin manner across back-end filesystem objects, while also sorting requests within batches according to their logical file or physical disk offsets. The main aim of this policy is to minimize costly disk drive head seek operations, in order to yield increased I/O throughput values.
[d] Target Round Robin (**TRR**); like ORR, this policy is only compatible with “ost_io” PTLRPC services on OSS nodes. It functions in the same manner as ORR, but batch-schedules OST_READ and OST_WRITE requests (along with offset ordering within request batches) across OSTs.

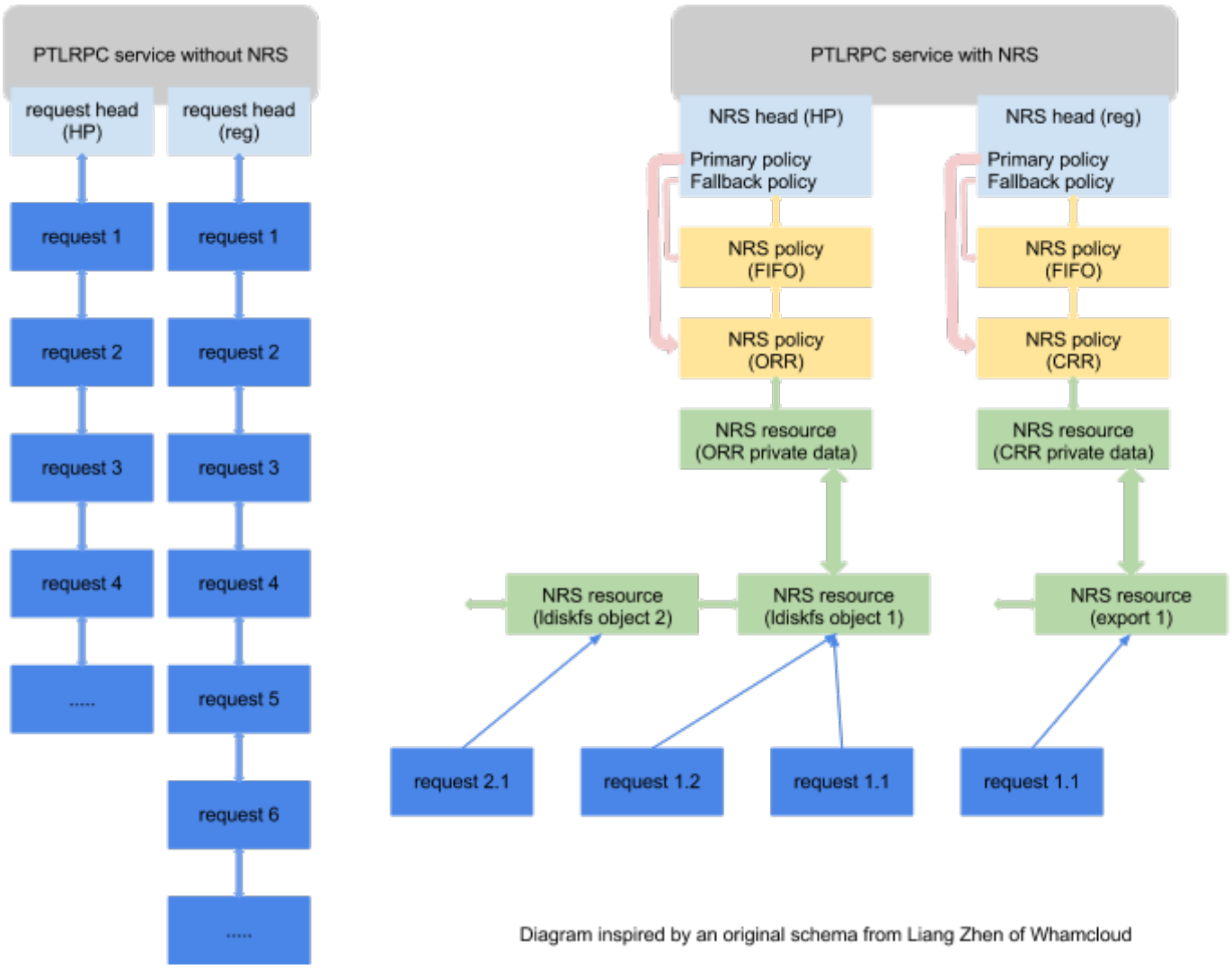
NRS resources (ptlrpc_nrs_resource) are various types of objects that are integral to the operation of the different policy types; they are implemented in a way that allows the manipulation of the relevant data structures in a generic manner by the NRS framework. NRS resources of a policy share a parent-child relationship, with NRS policy instances being the parents, and objects that take place in a prominent manner in the scheduling of requests in each policy, being the children. The concept is depicted in the figure below, for the CRR-N and ORR NRS scheduling policies, as an example.



NRS heads (ptlrpc_nrs) act as containers of policy instances that are compatible with a given PTLRPC service. Each PTLRPC service has at least one NRS head for policy instances that are used for handling regular priority requests, and optionally a second NRS head for policy instances handling high priority requests (if there is provision in the given PTLRPC service for handling high priority requests_.

All NRS heads use an instance of the FIFO policy as the *default/fallback* policy, along with an optional instance of a *primary* policy, which can be any of the aforementioned policy types (besides FIFO), as long as these are compatible with the given PTLRPC service and server type (e.g. an ORR policy instance is only applicable to ost_io PTLRPC services in OSS nodes, and in the future the use of some policies may be restricted on other PTLRPC services or server types). More than one primary-type policy can be associated with an NRS head, but only one can be acting as the enabled primary policy at a given point. All NRS heads make use of the fallback policy (FIFO) for handling all request types by default at PTLRPC service startup time, while primary policies can be enabled and disabled via interaction with lprocfs. In the future, more elaborate methods of switching between policies, or selecting the default startup policy for each PTLRPC service may be implemented.

The schema below that has been inspired by an original schema by Liang Zhen depicts the relationship between PTLRPC services and requests, and NRS heads, policies, and resources.



When no policy has been designated as the primary one in an NRS head, all requests for that service are handled by the default (FIFO) policy. In NRS heads with both a primary and a default policy, the primary policy is used in order to handle all incoming requests. The primary policy can then either deliberately or not, exhibit a failure in the handling of the request; deliberate failures can occur when the policy for some reason intentionally chooses not to handle the request (e.g. the ORR policy handles only OST_READ, and/or OST_WRITE requests, but other types of RPCs can also arrive at ost_io PTLRPC services, so ORR refuses to handle these unsupported types of requests); non-deliberate failures can occur due to errors in the operation of the primary policy (e.g. failure in allocating memory for an NRS resource instance). Requests whose handling by the primary policy failed, are then routed by the NRS framework to the default policy; default policy instances cannot fail in their operation (i.e. they handle all types of RPCs and do not perform operations that can fail at run-time), such that they are used as a safe path for handling requests.

2.2 Logic

NRS policies implement operations (ptlrpc_nrs_pol_ops) in the server request handling path that are invoked by the NRS framework; these can be viewed as hooks into previous non-NRS operation. The NRS framework and aforementioned policy operations collectively implement the following NRS request handling operations:
[a] **Start** (ptlrpc_nrs_req_start_nolock()), is invoked right before scheduling a request; its main purpose is to be used as a callback for resource and job control in the future.
[b] **Initialize** (ptlrpc_nrs_req_initialize()), obtains usage references on policies, allocates any resources required and obtains resource hierarchy handles for the request.
[c] **Enqueue** (ptlrpc_nrs_req_add()), adds the request to a policy's set of requests, first using the primary policy, if any, and then the fallback policy if the primary policy fails or does not exist.
[d] **Poll** (ptlrpc_nrs_req_pol()), obtains the request to be handled next for a given NRS head. All policies for an NRS head that have requests pending are queried in a round robin fashion in order to obtain the request. This mechanism could be made more elaborate in a future implementation if required.
[e] **Dequeue** (ptlrpc_nrs_req_del_nolock()), removes a request from a policy's set of requests; in the common case this is a request that was previously obtained for handling via a Poll operation, and has been handled to completion.
[f] **Finalize** (ptlrpc_nrs_req_finalize()), releases references on policies and references on resource hierarchies, and optionally the resources themselves, for a given request.
[g] **Stop** (ptlrpc_nrs_req_stop_nolock()), is invoked right after a request has been scheduled; like Start, its main purpose is to be used as a callback for resource and job control in the future.

These operations are carried out under existing PTLRPC lock protection where appropriate, which is handled by caller sites.

2.3 Binary Heap

Server nodes may at any point retain up to 1,000,000+ requests queued, waiting to be serviced. If NRS policies are to provide more elaborate schemes than the existing (non-NRS) FIFO ordering, there will need to be a method of maintaining prioritized queues of these large numbers of RPCs, so insertion and removal times at such large sets will have to be kept to a minimum if performance is to not suffer; this means making use of a highly scalable data structure.

To this end, the NRS task adds a binary heap implementation to libcfs; this is used to implement a priority queue ADT that can be used to implement NRS policies; preliminary testing has shown (source: Eric Barton/Liang Zhen) that this priority queue implementation exhibits prioritized insertion and removal times for an element in sets of 1,000,000+ elements, nearly in the sub-microsecond order. The important part of the binary heap API consists of the following operations:

[a] **Create** (cfs_binheap_create()**)**, for creating a binary heap that can accommodate a given number of initial elements, with the size being able to grow and shrink dynamically automatically as needed; the elements held in the heap are of type cfs_binheap_node_t.
[b] **Destroy** (cfs_binheap_destroy()), for deallocating all memory associated with the binary heap.
[c] **Find** (cfs_binheap_find()), for obtaining an element corresponding to a particular index, from the binary heap; this is usually used inside a wrapper operation, **Root** (cfs_binheap_root()), in order to obtain the element at index 0, which is the highest priority request, as this is the request most policies will want to obtain via Poll for handling at any given point. An important point to note is that there is no search operation, as this is rather expensive to perform on binary heaps, but for the reason just mentioned, this is not really necessary (i.e. the request at index 0 will be the next to serve at all times).
[d] **Insert** (cfs_binheap_insert()), for sort-inserting an element into a binary heap.
[e] **Remove** (cfs_binheap_remove()), for sort-removing an element from a binary heap.

Some wrapper functions are also part of the API, some of which may be useful to policies, depending on how they need to manipulate their binary heap instances, but discovering these is left up to the interested reader.

The heap will take care of keeping all of its elements sorted according to the results of relational comparisons between the heap's elements' keys; thus, users of the binary heap (policies) only need to provide a comparison function (cfs_binheap_ops_t::hop_compare()), which yields the direction of inequality between two heap elements; since the function is provided by the user, it can be arbitrarily complex, so different types of policies should be able to make use of this scheme, although it may have to be more elaborate for some.

Heap operations assume that callers handle all locking, and as long as this is true, the heap will atomically make sure that the lowest-key element is at the root of the heap tree (libcfs heap is an implementation of a min-heap); policies can make use of their internal data structures and/or information carried within each request in order to achieve the desired scheduling effect, by controlling the way in which relational inequality operations are evaluated, thus grouping RPCs as they please.